Higher-order frameworks for profiling and matching heterogeneous data

Simon Price
Department of Computer Science
University of Bristol

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Doctor of Philosophy in the Faculty of Engineering, Department of Computer Science.

June 2014

Word count: circa 70,000

Abstract

This Thesis brings together complementary research from higher-order computational logic and workflow systems to investigate software and theoretical frameworks for profiling and matching heterogeneous data. A motivating use case is submission sifting, which matches submitted conference or journal papers to potential peer reviewers based on the similarity between the paper's abstract and the reviewer's publications as found in online bibliographic databases. Inspired by e-Science workflows, we introduce the SubSift submission sifting framework for developing web-based research intelligence applications that profile and match heterogeneous textual content from web pages and documents. Abstracting SubSift we define a formal higher-order dataflow framework that ranges over a class of higher-order relations that are sufficiently expressive to represent a wide variety data types and structures. This dataflow model is shown to be embarrassingly parallel. JSONMatch, our proof of concept serial implementation, is used to demonstrate that the combination of this model and higher-order representation provides a flexible approach to analysing heterogeneous data. Finally we propose a theoretical framework for querying structured data, elevating Codd's relational algebra to a higher-order algebra defined on the basic terms of a higher-order logic. An extension incorporates approximate joins on structured data and is demonstrated to be feasible and have promise for future work.

Acknowledgements

Firstly, I would like to thank my co-authors of works cited in this Thesis: Christopher Bailey and Nikki Rogers (*ILRT, University of Bristol*), Peter Flach, Bruno Golénia and Sebastian Spiegler (*Intelligent Systems Laboratory, University of Bristol*), John Guiver and Thore Graepel (*Microsoft Research, Cambridge*), Ralf Herbrich (*Amazon, Mountain View, CA*), and Mohammed Zaki (*Rensselaer Polytechnic Institute*). Thanks are also due to further colleagues at ILRT for their expert advice on RDF and other topics: Mike Jones, Damian Steer and Jasper Tredgold. The implementation work in Chapter 5 benefited enormously from software engineering debates with my co-authors of, "*Coding guidelines for Prolog*": Michael Covington, Roberto Bagnara, Richard O'Keefe and Jan Wielemaker.

I am grateful to the following conference Programme Committee chairs for testing the SubSift software: Peter Flach (*KDD'09* and *ECML/PKDD'12*), Bart Goethals (*SDM'10*), Qiang Yang (*KDD'10*), Geoffrey Webb (*ICDM'10*) and Mohammed Zaki (*KDD'09* and *PAKDD'10*) – some of whom merit a second mention, in their capacity as journal Editor-in-Chiefs and Editorial Board members, for adopting SubSift in support of their reviewer assignment process: Peter Flach (*Machine Learning*), and Bart Goethals and Geoffrey Webb (*Data Mining and Knowledge Discovery*).

Special recognition should be given to my Thesis examiners, Carole Goble CBE (University of Manchester) and Tim Kovacs (University of Bristol), for their insightful reviews and a thoroughly inspirational viva. Finally, I am particularly indebted to Peter Flach, my supervisor and colleague, for his contagious passion for data science and his unwavering support.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE:.....

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Use Case 1 – Submission Sifting . . . 2

1.1.2 Use Case 2 – Finding an Expert . . . 3

1.1.3 Use Case 3 – Visualising Similarity Networks . . . 3

1.1.4 Use Case 4 – Profiling Reading Lists . . . 4

1.1.5 Use Case 5 – Ranking News Stories . . . 4

1.1.6 Related Use Cases . . . 5

1.2 Frameworks . . . 6

1.2.1 Dataflows . . . 6

1.2.2 Workflows . . . 10

1.2.3 Dataspaces . . . 18

1.3 Research Questions and Objectives . . . 20

1.4 Contributions . . . 22

1.5 Publications . . . 24

1.6 Structure of Thesis . . . 25

2 Background **27**

2.1 Terminology . . . 27

2.2 Informative Background Topics . . . 29

2.2.1 Representing Structured Data . . . 30

2.2.2 Comparing Heterogeneous Data . . . 40

2.2.3 Information Retrieval . . . 44

2.2.4 Comparing Structured Data . . . 45

2.3 Required Background Theory . . . 49

2.3.1 Vector Space Model . . . 49

2.3.2 Individuals as Terms . . . 50

2.3.3 Basic Terms in a Higher-Order Logic . . . 52

2.3.4 Kernels and Distances for Basic Terms . . . 57

2.3.5 RESTful Web Services . . . 61

2.4 Summary . . . 63

3 Workflows for Profiling and Matching Textual Content **65**
3.1 SubSift Case Study . . . 65
3.2 SubSift Web Services . . . 79
3.2.1 Example REST API Enactment of Submission Sifting . . . 82
3.3 Profiling and Matching Workflow Demonstrations . . . 86
3.3.1 Use Case 1 – Submission Sifting . . . 87
3.3.2 Use Case 2 – Finding an Expert . . . 88
3.3.3 Use Case 3 – Visualising Similarity Networks . . . 89
3.3.4 Use Case 4 – Profiling Reading Lists . . . 91
3.3.5 Use Case 5 – Ranking News Stories . . . 91
3.3.6 Related Use Cases . . . 94
3.4 Summary . . . 96

4 Higher-Order Dataflows **99**
4.1 A Higher-Order Dataflow Model . . . 101
4.1.1 Generator Function (g) . . . 103
4.1.2 Data Constructor (h) . . . 105
4.2 Knowledge Representation . . . 105
4.3 Formalising the Higher-Order Dataflow Model . . . 106
4.3.1 Paths . . . 111
4.3.2 Path Expressions . . . 113
4.3.3 Parallelisability and Scalability of the Model . . . 115
4.3.4 Continuations in the Model . . . 121
4.4 Implementation . . . 123
4.4.1 Data Formats . . . 123
4.4.2 Templates and Embedded Functions . . . 124
4.4.3 Example Dataflows . . . 125
4.4.4 Other Example Dataflows . . . 131
4.5 Comparison of JSONMatch with SubSift . . . 132
4.5.1 Submission Sifting Workflow Implementation . . . 132
4.5.2 Asynchronous Behaviour . . . 134
4.5.3 Feature Comparison . . . 135
4.5.4 Summary of Comparision . . . 138
4.6 Summary . . . 138

5 Querying and Merging Heterogeneous Data **141**
5.1 Relational Model . . . 143
5.1.1 Relational Operations . . . 144
5.1.2 Relational Algebra . . . 148
5.2 Lifting the Relational Model to a Higher-Order . . . 148
5.2.1 Indexing Basic Terms . . . 149
5.3 Basic Term Relational Model . . . 157

5.3.1 Basic Term Relational Representation . . . 157
5.3.2 Indexing Basic Term Relations . . . 158
5.3.3 Basic Term Relational Operations . . . 158
5.3.4 Basic Term Relational Algebra . . . 161
5.4 Approximate Relational Joins . . . 162
5.4.1 Proximity-Joins . . . 162
5.4.2 Basic Term Proximity-Join . . . 163
5.5 Proof of Concept . . . 164
5.5.1 Kernels and Distances for Approximate Joins . . . 164
5.5.2 Demonstrations . . . 168
5.6 Summary . . . 173

6 Related Work **179**
6.1 Broadly Related Areas . . . 179
6.1.1 Big Data . . . 180
6.1.2 Clustering . . . 181
6.1.3 Expert Finding . . . 181
6.1.4 Higher-Order Frameworks . . . 182
6.1.5 Scientometrics and Bibliometrics . . . 183
6.1.6 Semantic Web . . . 183
6.1.7 Text Processing . . . 184
6.2 Approaches to Profiling and Matching . . . 185
6.2.1 Schema and Ontology Matching . . . 185
6.2.2 Dataspaces . . . 191
6.2.3 Discussion of Profiling and Matching Approaches . . . 191
6.3 Comparisons with Other Frameworks . . . 193
6.3.1 Comparison with Other Formalisms . . . 193
6.3.2 Comparison with Other Software Frameworks . . . 194
6.4 Scope . . . 196
6.5 Summary . . . 197

7 Discussion **199**
7.1 Review of Research Objectives . . . 199
7.2 Limitations . . . 200
7.2.1 Use of hard-coded algorithms . . . 200
7.2.2 Inefficiency of naive HTTP-based workflow enactment . . . 202
7.2.3 Homepage structural heterogeneity . . . 203
7.2.4 Impact of component substitution . . . 205
7.3 Impact . . . 207
7.3.1 Academic Peer Review . . . 207
7.3.2 Research and Education . . . 209
7.3.3 Applied Research Projects . . . 210

7.4 Summary 211

8 Conclusions and Future Work **213**
8.1 Conclusions 213
8.2 Future Work 215
8.2.1 Enhancements to Submission Sifting Workflows 215
8.2.2 Scaling-up to *Big Volume* Big Data 216
8.2.3 Higher-Order Relational Database 216
8.2.4 Social Network Analysis 217
8.2.5 Research Objects 217
8.2.6 Virtual APIs 220
8.2.7 Summary of Future Work 220

Appendices **223**

A A JSONPath Primer **223**
A.1 JSON 223
A.2 JSONPath 224

B JSONMatch Functions **229**
B.1 External Extension Functions 230
B.2 Internal Extension Functions 232
B.3 Built-in Functions 233

C JSONMatch Submission Sifting Workflow **237**
C.1 Profiling Submitted Papers 238
C.2 Profiling Programme Committee (PC) Members 245
C.3 Profile Matching 249
C.4 Bid Initialisation 252

List of Tables

2.1 EXAMPLE FS PARAMETERS ADAPTED FROM [Jar89] 43
2.2 INTERPRETATION OF HTTP REQUEST METHODS IN REST . . . 62

4.1 EXAMPLE PATHS 113
4.2 EXAMPLE PATH EXPRESSIONS 116
4.3 EXAMPLE JSONMATCH FUNCTIONS 125
4.4 JSONMATCH AND SUBSIFT COMPARISON 136

7.1 EXAMINER EXTENSIONS TO SUBSIFT WEB HARVESTER. 204

A.1 EXAMPLE JSONPATH ABSOLUTE EXPRESSIONS 225
A.2 JSONPATH SYNTAX ELEMENTS [Gös07] 226
A.3 EXAMPLE JSONPATH EXPRESSIONS 227

B.1 JSONMATCH META FUNCTIONS 233
B.2 JSONMATCH WEB FUNCTIONS 233
B.3 JSONMATCH TIME FUNCTIONS 233
B.4 JSONMATCH SET FUNCTIONS 234
B.5 JSONMATCH ARRAY FUNCTIONS 234
B.6 JSONMATCH STRING FUNCTIONS 235
B.7 JSONMATCH TEXT FUNCTIONS 235
B.8 JSONMATCH COMPARISON FUNCTIONS 235

List of Figures

1.1 Dataflow diagram. 7
1.2 Control flow diagram. 7
1.3 Distribution of workflow types in myExperiment repository. 12
1.4 Galaxy genomics workflow. 13
1.5 Taverna workflow. 14
1.6 RapidMiner workflow. 15
1.7 Illustrative example of an *abstract workflow diagram*. 16

2.1 Tabular representation of the person and town relations. 33
2.2 Adjacency list representation of a tree. 34
2.3 Nested sets representation of a tree. 34
2.4 Materialised paths representation of a tree. 35
2.5 Lists $[A,B,C]$ and $[A,D]$ as basic structures. 54
2.6 B-tree as a basic structure. 55

3.1 Enter Names step of Reviewer Profile Builder in SubSift prototype. . 73
3.2 View Reports step of Profile Matcher in SubSift prototype. 74
3.3 Top-level menu depicting workflow of SubSift prototype. 75
3.4 Generalised Submission Sifting Workflow. 80
3.5 SubSift System Architecture. 81
3.6 Generalised submission sifting enacted using SubSift REST API. . . 83
3.7 SubSift personalised report listing top ranked papers. 86
3.8 Bottom section of a personalised report showing PC member profile. 86
3.9 Schematic of *submission sifting* workflow. 87
3.10 Web 2.0 wizard implementation of the *submission sifting* workflow. 88
3.11 Similarity match between text fragment and ILRT staff. 89
3.12 Term contributions for matches between text and ILRT staff. 90
3.13 Schematic of *Finding an Expert* workflow. 90
3.14 ILRT staff homepage similarity match graph visualisations. 91
3.15 Schematic of *Visualising Similarity* workflow. 92
3.16 Tag cloud visualisation of researcher interests. 92
3.17 Schematic of *Profiling Reading Lists* workflow. 93

3.18 Schematic of *Ranking News Stories* workflow. 94
3.19 *Find Similar Research* example. 96
3.20 *Profile Email Recipients* example. 97

4.1 Transformation $\Phi(g)(h)(s,t,u,\dots) = v$. 102
4.2 Dataflow $\Phi_3(\Phi_1(s),\Phi_2(t)) = w$. 102
4.3 Dataflow $\Phi_5(\Phi_4(\Phi_1(s)),\Phi_4(\Phi_2(s)),\Phi_4(\Phi_3(s))) = z$. 103
4.4 Three kinds of higher-order *map* transformations. 104
4.5 Higher-order *product* transformation $\Phi(\times)(h)(s,t) = v$. 104
4.6 Higher-order *lambda* transformation $\Phi(\lambda)(h)(s) = v$. 105
4.7 Basic relation. 107
4.8 Map transformation of basic relations. 109
4.9 Product transformation of basic relations. 109
4.10 Lambda transformation of basic relations. 110
4.11 Embarrassingly parallel functions and pure transformations. 119
4.12 Different execution strategies. 122
4.13 PC profiling part of *submission sifting* workflow. 127
4.14 JSONMatch implementation of PC profiling. 128
4.15 JSONMatch implementation of *submission sifting* workflow. 134

5.1 Term-based indexing for basic structures. 151
5.2 Term-based indexing for basic abstractions. 151
5.3 Type-based indexing for basic tuples. 155
5.4 Type-based indexing for basic structures. 156
5.5 Type-based indexing for basic abstractions. 157
5.6 Type name-based and type-based indexing for type *Author*. 158
5.7 Dendrogram of proximity-joins on the CORA publication type. . . . 175
5.8 Precision and recall on the CORA publication type. 176
5.9 Basic term proximity-join as a SubSift report. 177

7.1 Yahoo! Pipes designed as data sources for SubSift. 206
7.2 Pipe to restrict papers retrieved from DBLP by their age. 206
7.3 Example results from Machine Learning journal Matcher tool. . . . 208
7.4 Machine Learning journal Editorial Board membership editor. . . . 209
7.5 ECML-PKDD'12 conference planner iPhone app. 212

8.1 Using SubSift similarity data to cluster ILRT staff homepages. . . . 218
8.2 Pairwise precision and recall for clustered ILRT staff homepages. . . 219

C.1 Higher-order transformations in the *submission sifting* workflow. . . 237

Chapter 1

Introduction

This Thesis is concerned with *frameworks*: software and theoretical frameworks for profiling and matching heterogeneous data. We develop and investigate frameworks in the context of 'research intelligence' applications involving data about researchers and their organisations. Our work brings together the otherwise disparate research traditions of *higher-order computational logic* and *workflow systems*; as such, this Thesis constitutes a bridge between these communities and serves to demonstrate their joint potential for addressing software engineering problems in this increasingly important domain.

In this first chapter we introduce a range of research intelligence use cases that motivate our work, outline the scope of the software architectures and theoretical foundations from which our frameworks emerge, introduce our research questions and objectives, highlight the contribution to knowledge, list a number of supporting peer-reviewed publications and outline the organisation of the rest of the Thesis.

1.1 Motivation

Our work is motivated by a rich vein of research intelligence use cases based around web applications that support the academic research process, its researchers and their organisations by profiling and matching latent information buried in extant heterogeneous data. The web applications, in and of themselves, are not new but have historically been produced as individual, standalone, purpose-written applications despite the fact that they have many engineering and data features, and hence problems, in common. To frame these problems and highlight the research questions that flow from their consideration, we introduce a range of motivating use cases. All these use cases exploit the intuitive idea that the published works of researchers, research groups and organisations, in some sense, describe their specific research interests and expertise. By analysing these published works in relation to the body as a whole, discriminating profiles may be produced that effectively characterise heterogeneous documents ranging from traditional academic papers to web sites, blog posts and Twitter feeds. Such profiles have applications in their own right but can also be used to compare one body of documents to another, ranking arbitrary combinations of documents and, by proxy, individuals or groups by their similarity to each other. This idea may be further extended by relaxing the definition of a 'document' to include

other data sources, such as records held in organisational databases, queries to online repositories, browser bookmarks, email, Linked Data, the Semantic Web, and a multitude of other relevant data sources.

The principle of using existing data to drive these web applications avoids researchers having to spend their time manually defining and maintaining their own research profiles by providing distilled information about their research interests, works and personal preferences – potentially to multiple research intelligence applications. Such manually supplied summary information, keywords being an obvious example, is often highly subjective and an unreliable method of labelling [BDK+04, BJZ02]. The web applications described in the use cases therefore assume some means of computing a profile from existing data sources. Additionally, if multiple web applications are able to access these profiles, or be able to recreate exactly the same profiles given the same data, then they not only avoid wasting researchers' time but also simplify direct comparison and matching of profiles.

The following motivating use cases are described below and are referred to throughout the Thesis.

Use Case 1 – Submission Sifting

Use Case 2 – Finding an Expert

Use Case 3 – Visualising Similarity Networks

Use Case 4 – Profiling Reading Lists

Use Case 5 – Ranking News Stories

To emphasise the potential re-use of these use cases, we also briefly describe a number of closely related use cases concerned with mining and mapping the research landscape of a large research university.

1.1.1 Use Case 1 – Submission Sifting

Peer review of written works is an essential pillar of the academic research process, providing the central quality control and feedback mechanism for submissions to conferences, journals and funding bodies across a wide range of disciplines. However, from the perspective of a busy conference chair, journal editor or funding manager, identifying the most appropriate reviewer for a given submission is a non-trivial and time-consuming task. Effective assignment, first and foremost, requires a good match to be made between the subject of the submission and the corresponding expertise of reviewers drawn from a, sometimes large, pool of potential reviewers. In the case of conferences, a recent trend transfers much of this allocation work to the reviewers themselves, giving them access to the full range of submissions and asking them to bid on submissions they would like to review. Their bids are then compared to inform the allocation decisions of the programme committee chair. This *submission sifting* use case covers three specific stages in the conference peer review process:

1. matching submissions to reviewers;
2. ranking potential assignments;
3. allocating papers to reviewers.

In the first step, each reviewer's bids are initialised based on textual similarity between the text of the paper and the text of reviewer's publications, as listed in a bibliographic database or on their web homepage. In the second step, each of the reviewers is provided with a personalised web page listing details of all papers ordered by initial bid allocation and similarity to their own published works. Guided by this personalised perspective, plus the usual titles and abstracts, reviewers affirm or revise their bids. In the final step, after all reviewer bids are submitted the programme chair consults, for any paper, a similarity ranked list of reviewers to assist them in allocating papers with either too few or too many bids. Applications addressing this use case can be integrated components of conference management systems[1][2] or as standalone web applications [HP06, MM07, FSG+09, Abb11, CZB11, Pen11].

1.1.2 Use Case 2 – Finding an Expert

Finding an expert on a given topic within an organisation can be difficult, especially if the person searching is not familiar with all members of that organisation or with the topic itself. This is often true for members of the media seeking an expert, within a university, on some newsworthy scientific report. It is also true for journal editors deciding which editorial board members to ask to review a particular paper.

Both variants of this problem can be captured by a single use case. In this use case the user submits a fragment of text, typically an abstract or the full text of a paper, and the application compares this against the home pages or publications of a pre-defined list of researchers. The result is a list of researchers ranked by their similarity to the submitted fragment. The results are displayed as a ranked list or visualised as, for instance, a bar chart. Optionally, the list of terms contributing to each researcher's similarity score can be viewed along with the percentage that each term contributed to the combined score. Expert finding in both research intelligence and business intelligence domains has a long and established body of prior work to inform applications in this use case [YsK03, BAdR06, FZ07, MM07, DKL08].

In many ways, expert finding can be viewed as a special case of the submission sifting use case but where the emphasis is on finding the best match for a single submission rather than for multiple submissions. However, submission sifting tends to be a 'one shot' use for a specific event whereas an application for finding an expert will be used over an extended period of time and thus must also consider strategies for updating profiles to keep them current.

1.1.3 Use Case 3 – Visualising Similarity Networks

Organisations usually have a formal hierarchical structure but there can also be hidden structures, both of which can potentially be discovered by analysing the similarity

[1] EasyChair – `http://www.easychair.org`, visited May 2014.
[2] VIVIWeb Searchlight – `http://about.vivosearchlight.org`, visited May 2014.

of documents associated with members of the organisation. The similarity between individuals is represented as an undirected labelled graph where the vertices are the individuals and the edges the similarity relation between two individuals. One strategy for making structure visible is to only add edges between nodes whose similarity score is above a threshold. Another is to only add the top n edges for each node, where n is a small integer. The resultant similarity graph is rendered as a static (or interactive) image, optionally labelling the edges with their associated similarity scores and contributing terms.

In a variation of this use case, a strong assumption of the existence of an hierarchical structure is made in the construction of the similarity graph such that a tree or dendrogram is assembled as the visualisation.

1.1.4 Use Case 4 – Profiling Reading Lists

Many researchers share lists of web page bookmarks (URLs) to social bookmarking websites like Bibsonomy, CiteULike and Delicious [BHJ+10, LB10, SAYMY08]. As their interests change over time, so does the list of bookmarks. The bookmarks and the text of the web pages that they link to can be used to create a profile of a researcher's current interests.

A profile constructed from social bookmarks could, in principle, be used in Use Case 1, but in practice the level of adoption of such bookmarking sites by reviewers is not yet sufficiently high for this method to be used alone. Instead, in this use case for those researchers who do use social bookmarking, the researcher pastes the url of their social bookmarks webpage into the application which then produces a summary list of keywords derived from the linked webpages. The list is ranked according to the relative importance of keywords within the overall corpus. Optionally, the list is visualised as a word cloud, in which text size is proportional to keyword importance. The advantage of periodically generating this list automatically from normal research activities is that it is self-updating.

Due to pre-Open Access legal restrictions, this use case excludes the potentially more widely useful idea of profiling from links to online papers in journals and proceedings. However, leaving aside legal issues, the principle is exactly the same as for profiling social bookmarked webpages and so this use case could be extended to cover such papers in a future Open Access scenario.

1.1.5 Use Case 5 – Ranking News Stories

A researcher's subscriptions to RSS blog and news feeds provide frequently updated textual information. However, reading RSS feeds can be time consuming and what may initially appear relevant from the title may turn out not to be after following the associated link to the full text. The ability to rank stories in a feed by relevance to the research interests of the reader helps to avoid missing potentially relevant articles and is a potential time saver, reducing the amount of information the researcher needs to scan.

1.1.6 Related Use Cases

The University of Bristol, in common with many large research organisations, has a substantial body of researchers and a commensurately large portfolio of research projects spanning subjects from archaeology through to zoology. Historically at Bristol, as elsewhere, researchers and their projects are organised into departments, faculties, schools and so on, largely on the basis of their discipline. An unfortunate side-effect of this organisational structure is that researchers tend to know of other researchers and projects within their own branch of the organisational hierarchy. It is not untrue to say that they know more about researchers and projects elsewhere around the world than in their own institutions. This can leave researchers unaware of potentially relevant research going on elsewhere within their own university. Recognising that many important research areas now span multiple disciplines, the University Research Committee identified a need to produce research intelligence tools to mine and map its research landscape. The aim being to providing its researchers and its Research and Enterprise Development group with new ways of searching, accessing and visualising connections between existing research – irrespective of the organisational boundaries and structures.

This requirement generated a number of variations and enhancements of the previously described use cases. We list them below and note that, even where use cases are otherwise similar to their earlier versions, the data sources involved are an order of magnitude larger and contain data from internal corporate databases as well as external websites.

- **Find a Researcher**. The user pastes some text into a web form, for example a few keywords or the abstract of a paper, and the application finds the most similar researchers in a selected school or department.

- **Find Similar Research**. The user defines the research they would like to find by browsing through and selecting examples from a list of schools, departments and researchers, and the application finds similar examples.

- **Find Research Networks**. The user selects one or more schools or departments and the application cross-matches all their researchers to find networks of similar research, displaying them as a similarity network diagram.

- **Profile Email Recipients**. The user pastes a list of University of Bristol email addresses into a web form and the application displays each person's full name, job title, contact details, homepage url and research profiles, optionally also matching them to find networks of similar research.

The first of these related use cases closely resembles Use Case 1 and is designed to be part of the public-facing website as an alternative to the current manually curated Directory of Experts. The second two are mainly intended for use by researchers and by the research development group. The fourth use case is designed to be used by recipients of invitation emails for large internal multidisciplinary events, enabling them to profile and investigate connections between the invitees.

1.2 Frameworks

We use the term *framework* in this Thesis to refer to either:

- formal theory that guides the design of some class of similar applications, or
- software architecture and toolkit for the implementation of such applications.

Benefits sought through the creation and use of frameworks include the encapsulation of design concepts and re-usable functionality in a form that can be both readily understood and economically applied to new problems in the same class.

In this section we outline the scope of the software architectures and theoretical foundations from which the frameworks developed in our work emerges. We cover three main topics that correspond to the three core chapters of the Thesis: dataflows (Chapter 4), workflows (Chapter 3) and dataspaces (Chapter 5). The order of running here differs from the order of the chapters, but we introduce dataflows ahead of workflows because the latter are more easily explained using the former.

1.2.1 Dataflows

As its name suggests, *dataflow* represents the flow of data through a system. In software systems design, dataflow is often depicted and communicated to developers and other stakeholders using a dataflow diagram, such as the one in Figure 1.1. The interconnections and dependencies between components in a dataflow can also be encoded, either manually or with tool support, as an executable version of the described system. By contrast, the related but different concept of *control flow* explicitly represents the sequence of execution steps in a system and similarly can be depicted as a diagram, such as the flow chart in Figure 1.2. The term *flow* in 'flow chart' refers to the flow of control in a system, not the flow of data.

In terms of execution, the important difference dataflow and control flow is that dataflow does not necessarily specify the sequence of execution: dataflow is driven by dependencies between components of the system communicated by shared variables or message passing so that a component will execute as soon as all its inputs are available. In the dataflows involving multiple independent components, it is possible for work to be done in parallel. Under such circumstances, the control flow of the dataflow may be non-deterministic and unpredictable, but still guaranteed to produce the expected behaviour. Ensuring that an implementation of a dataflow design honours that guarantee can be challenging if there are complex networks of dependencies that might, for instance, give rise to deadlocks. This is one of the reasons why there have been a number of efforts to formalise dataflows and construct them on firm theoretical foundations. Another reason, more important in our own motivation for pursuing dataflow formalism, is that the existence of a formal description of a dataflow system (referred to as a *dataflow model*) enables discussion and reasoning about the characteristics of physical systems based on the model ahead of their implementation. Such formal models offer a means to compare characteristics across different possible models prior to an implementation and also provide a 'recipe' for building systems to achieve those characteristics.

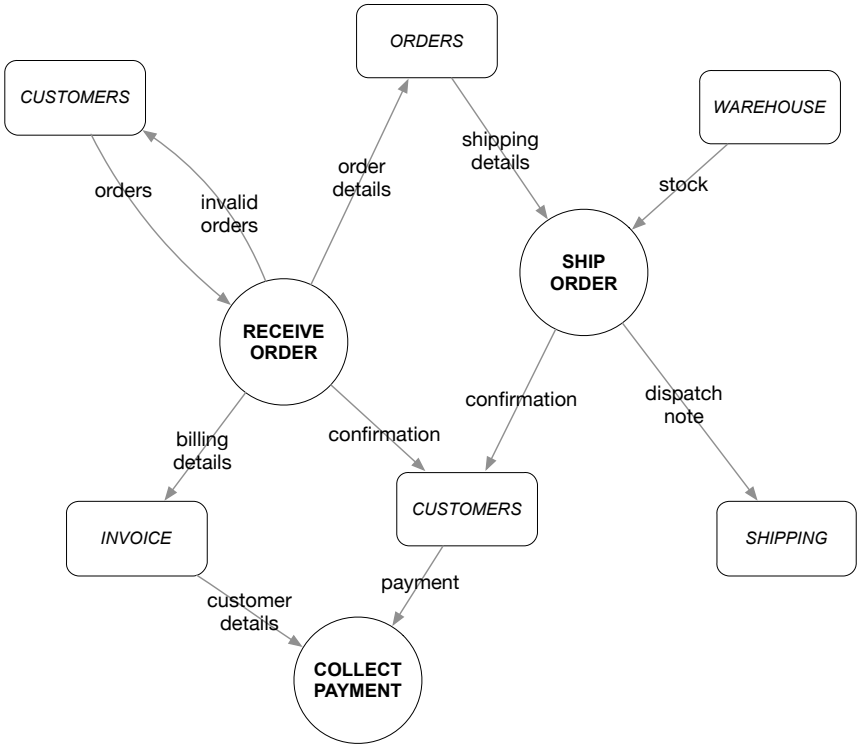


Figure 1.1: Dataflow diagram for an online book store. Circles represent processes, rectangles represent data sources, and lines represent data inputs and outputs.

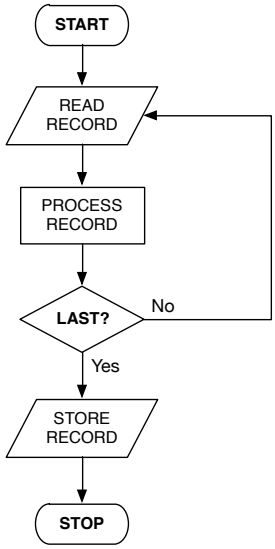


Figure 1.2: Control flow diagram depicting iteration over a set of records. Symbols represent steps in the execution. Diamonds represent decisions.

One of the oldest and most pervasive dataflow models is based on the concept of a linear chain of components, known as a *pipe* or *pipeline*, through which data flows sequentially from component to component[3]. This model is embodied in the familiar idiom of Unix pipes [Rit79], which take the output from one program and feed it in as input to the next program in the sequence, and so on, one program after the other until the last program outputs the final result. The Unix pipe operator '|' is often used in combination with a range of redirect operators to change the input and output behaviour of the overall pipe or of individual programs in the pipe, e.g. '<' to read a file from disk and '>' to redirect output to a file on disk. Unix pipes are a simple but powerful language implementing the pipeline dataflow model[4]. The general sytax for Unix pipes is:

```
% command_1 | command_2[ | command_3 ...]
```

As an example, the following Unix command is a pipeline of four programs: `curl`, `grep`, `sort` and `less`). The first command will fetch the HTML text of the University of Bristol homepage, pass this text to `grep` to filter and pass through only those lines that contain the string "bristol" onwards to the `sort` command, which in turn feeds its output to the text viewer called `less`.

```
% curl http://www.bris.ac.uk | grep "bristol" | sort | less
```

In Unix pipelines and for pipelines in general, it is possible to create dataflows that fork into separate branches, potentially with the branches rejoining later, thereby enabling the definition of graph topologies, as depicted earlier in Figure 1.1. With this informal intuition of a pipeline in mind, we now prepare for our introduction to dataflow formalisms by defining the concept of *dataflow* as follows.

Definition 1.2.1 (Dataflow) Dataflow *is a pipe that transports a set of values from a source to a destination. A dataflow represents a dependency of the destination on the source.*

Historically, dataflow has often been defined as transporting a *sequence* of values; we define it in terms of transporting a *set* of values to de-emphasise sequential delivery of values, thereby accommodating modern distributed parallel dataflow computation, typified by MapReduce [DG08]. In Chapter 4 we give our own formal definition of this pipeline behaviour using function composition (Section 4.1, p.101) in a higher-order logic, but here we introduce *monads* the most commonly used formal framework for describing the behaviour of pipelines.

Monads are a mathematically elegant way of adding a range of programming concepts, such as sides-effects, variable assignment, global variables, input/output, and exception handling, as behaviours associated with each component (function)

[3]Pipes were proposed by Doug McIlroy as far back as 1964, long before the advent of Unix, "We should have some ways of coupling programs like garden hose – screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.". `http://doc.cat-v.org/unix/pipes/`, visited May 2014.

[4]We note that dataflows with branching and parallel execution are also possible using Unix pipes.

in a pipeline – without otherwise affecting the intended behaviour of the function [Wad90]. For example, a 'global variables monad', M, represents the programming concept of global variables as an abstract structure that can be attached to functions in the pipeline. This is sometimes referred to as decorating the function with M. So, a function f that maps values of type a to values of type b, denoted $f : a \rightarrow b$, when decorated in such a way becomes $M\ f : a \rightarrow M\ b$, still having the same input and output behaviour but, by virtue of M, the monadic function '$M\ f$' can now use the programming concept of global variables without needing to change the behaviour of f itself. More formally, a *monad* is a triple $(M, unit, \star)$, known as a Kleisli triple, with parts defined as follows.

- M is a type constructor defined such that, for instance, given a type Int, the type constructor M builds a new type $M\ Int$. The intended meaning of 'type' is a set, e.g. Int means the integers, $\mathbb{Z}$. A 'type constructor' builds new types from old types, e.g. $List\ Int$ for type constructor $List$ constructs the type of lists of integers; similarly $List\ String$ constructs the type of lists of strings.

- $unit$ is a function $unit : a \rightarrow M\ a$, that maps a value in some given type a to a value in the monadic type $M\ a$. In other words, $unit$ puts a value from type a into a monadic container of type $M\ a$.

- $\star$ is a function $\star : (M\ a) \rightarrow (a \rightarrow M\ b) \rightarrow (M\ b)$, that chains a monadic value of type $M\ a$ with a function of type $a \rightarrow M\ b$ to create a monadic value of type $M\ b$.

Using monads, a wide range of real-world programming concepts can be cleanly embedded in an existing type system. In the context of pipelines this means that information (state) to be carried from component to component in the dataflow without requiring changes to the components themselves. Further details and a formal definition of monads, with examples for some of the aforementioned programming concepts, are given in [Wad90]. However, for readers just wanting an intuition of how monads are used, it may help to know that they are often referred to by programmers as "executable semi-colons", named after the semi-colon at the end of statements in the C family of programming languages and with the meaning that monads represent a declarative way of associating code with the end of each statement in a program[5].

Monads have been applied to other formalisms to cleanly define new formalism frameworks for representing dataflows. A noteworthy example of this approach uses monads embedded into the type system of computational lambda calculus [Mog89, Mog91], thereby augmenting the calculus to add computations defined as sequences of steps. The resultant formalism is used to define the dataflow model underpinning the Taverna workflow system [TMG+07]. Interestingly, this formalism applied retrospectively to formalise Taverna's syntax and semantics but the model turned out to be a remarkably close match to the, independently arrived at, implementation of the software framework.

Other approaches use contrasting formalisms to define their theoretical frameworks to describe dataflows, sometimes underpinning a real implementation and

[5] Monads are related to the concept of an *aspect* in aspect-oriented programming (AOP).

sometimes just as pure theory. An example of the latter uses nested relational calculus augmented with service functions to ensure that dataflows involving heterogeneous data in a database are well-defined [BNTW95]. That particular approach advocates relational calculus as a superior alternative to extensions to first-order logic or higher-order logic (as used in this Thesis) – a topic we revist in "Chapter 6 – Related Work". A different, earlier approach uses declaritive design and programming constructs based on process networks [KM77] to model the behaviour of the Kepler workflow system [ABJ + 04].

At a higher level of abstraction, [HKS + 07] draws on a range or prior formalisation efforts to define a formal model, not of a specific dataflow but for a repository of dataflows – reinforcing the point that the existence of a formal model underpinning specific dataflow implementations enables unambiguous mapping between them and facilitates the creation of dataflow repositories. Also at the repository level of formalisation, dataflow formalisms have been proposed that operate on collections in a repository, an activity known as collection-oriented dataflow programming, using pipelining of nested collections [MB05].

The formalisations described in this section are often closely linked to applications in scientific workflows, an area of e-Science. However, dataflow formalisation work has also been undertaken to underpin business workflows, most notably using petri nets to formalise the business workflow standard BPEL [OVvdA + 07]. Closing the current section and moving on to introduce workflows in the next section, we repeat an interesting observation from [LAB + 06] about the difference between business and scientific workflows: scientific workflows are closer to dataflows and usually resemble or actually implement dataflow process networks [KM77], whereas business workflows are control flow and task-oriented. In this Thesis, our workflows introduced in "Chapter 3 – Workflows for Profiling and Matching Textual Content" were inspired by prior work in e-Science and consequently depend on dataflow rather than control flow; their underlying dataflows play an important role in this Thesis and are the subject of "Chapter 4 – Higher-Order Dataflows".

1.2.2 Workflows

Although a relatively recent term, 'workflow' refers to an idea that is as old as computer science itself: the idea of decomposing software into re-usable parts (referred to as *components*) and controlling their invocation through another program. Indeed, workflows and components may be implemented as traditional piped commands, shell scripts, batch files, and in numerous other programming languages or environments. With this in mind, we formally define *workflow* as follows.

Definition 1.2.2 (Workflow) Workflow *is a directed graph where vertices represent activities and edges represent dataflow between activities. Activities (also referred to as* components*) are units of execution and may themselves be workflows. All workflows have one or more input ports and one or more output ports. Workflow enactment is the interpretation of the workflow graph by software (known as a workflow engine), executing activities in the required sequence to produce the described dataflow.*

Enactment can be effected through such code or under the control of Make, Ant, Maven, job schedulers, or purpose-designed workflow enactment systems that inter-

pret, for instance, XML or JSON abstract representations of a workflow. It is tempting, particularly for abstract XML or JSON representations, to say that workflows are different because they aspire to system independence. However, the same could be said of a variety of enactment schemes: for instance GXP Make has been used as a workflow enactment system with an inherently system-portable representation [TMM+13]. The aspect of workflows that is most relevant to their enactment is that they have an underlying dataflow which may be formally described as a model of the physical system. As we discussed in Section 1.2.1, opportunities for parallelism in a workflow can be inferred from dataflow dependencies between components. Consequently the order of enactment of components in a workflow involving parallelism can be non-deterministic, with components executing as soon as all their inputs are available. Further properties of a workflow's dataflow can be used to reason about the resource requirements of their enactment, for example whether a model is scalable to large datasets, a topic we explore in "Chapter 4 – Higher-Order Dataflows".

One of the advantages of enacting workflows through purpose-designed workflow engines is that they provide a level of architectural abstraction away from the physical infrastructure implementing individual components in the workflow. With such abstract descriptions of workflows it becomes easy (as compared to scripting an equivalent behaviour by hand) to substitute components for functionally equivalent ones or to invoke components remotely on other systems. Systems architectures that separate functionity into a 'storage and computational' layer, a 'logic' layer, and a 'user interface' layer are referred to as *three-tier architectures*. The components of a workflow correspond to the storage and computational layer; the workflow itself corresponds to the logic layer; and the web application from our use cases corresponds to the user interface layer. Remote components are often referred to as *services* and if the communication with the services (i.e. between the first two layers) takes place over the web it is known as a *web service*. The web services that control enactment of the workflows introduced in later chapters of this Thesis are discussed at the end of the current section and explained further in "Chapter 2 – Background". These services communication between the second two layers)

As with the idea of workflows themselves, the idea of implementing components using web service calls to remote systems is not unique to workflow systems. So, although workflow engines typically enable the invocation of components using web services, there is an argument that they are just another example of conventional *service-oriented architectures* (SOA) used throughout modern enterprise systems. Architecturally, that is true. However, unlike internal corporate enterprise development, academic research is an open social collaboration that relies on communication of ideas between researchers – many of whom have no formal background in software engineering. The importance of openness and clarity of workflows to academic peer review and the reproducibility of research was emphasised by the high-profile refutation of a significant scientific result in a top ranking journal [CWB07]. In the context of e-Science and *research data management* in general, the communication value of workflows may ultimately be as important as their enactment. For this reason, regardless of their underlying representation or enactment system, research workflows are commonly published with an accompanying *workflow diagram* to aid communication between researchers.

There are numerous system-specific and domain-specific workflow diagramming conventions and no single agreed standard for research workflows. This is in contrast to business process workflows, where considerable progress towards standardisation has been made across vendors and coalitions involving bodies such as ISO, OMG and W3C. While some work has been done on using business workflows in an e-Science setting [SKD10], as was mentioned at the close of Section 1.2.1, research workflow systems are usually viewed in terms of dataflow whereas business process workflows tend to emphasise control flow. Some measure of the range of research workflows in circulation can be gained from the leading domain-independent workflow repository, myExperiment [DRGS09]. At the time of writing, the myExperiment workflow repository and social network website holds over 2,500 workflows (August 2013). myExperiment has been operating long enough to have engendered some fairly detailed and in-depth analysis of the types of workflows it holds [BRGC + 12, DBL12, GAB + 12, LRL + 12]. However, here we only aim to convey the range of workflow systems represented. Figure 1.3 shows the distribution of workflow types in myExperiment, revealing that Taverna [HWS + 06, OGA + 06] and RapidMiner diagrammatic workflows dominate this multidisciplinary repository. RapidMiner is based on the Weka [BFH + 10] machine learning and data mining framework. Taverna, RapidMiner and Kepler [ABJ + 04] are not domain-specific in their use. Popular domain-specific diagrammatic workflows, such as Galaxy [GNT10] for genomics, are under-represented in myExperiment. In Galaxy's case this is probably because Galaxy incorporates its own built-in workflow social repository.

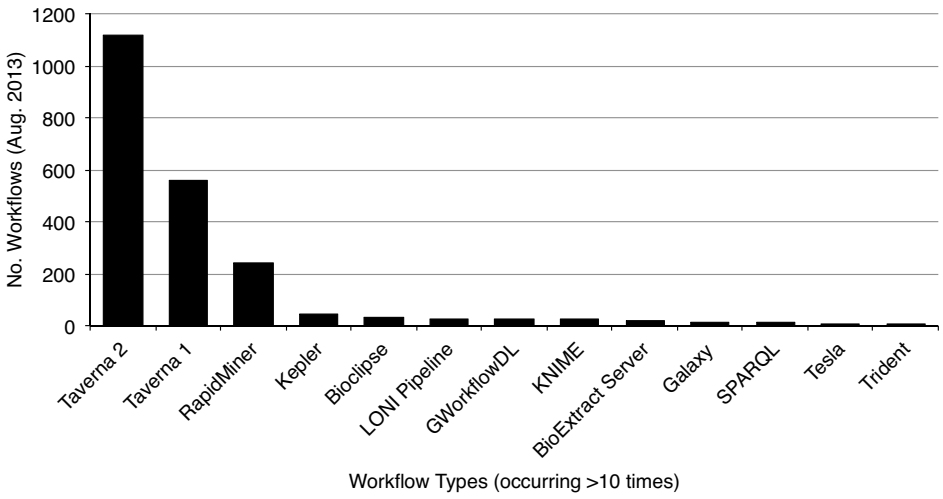


Figure 1.3: Distribution of workflow types in myExperiment repository.

Workflow diagrams are conceptual-level, graphical representations of the dataflow between the components of a workflow. For the use cases in this Thesis we wish to discuss workflows independent of any particular diagramming scheme and so, later, we will define an abstract workflow diagram representation. Before doing so, we describe three contrasting examples of workflows from the myExperiment repository to give a flavour of some popular workflow diagramming styles. Example 1.2.3 is a

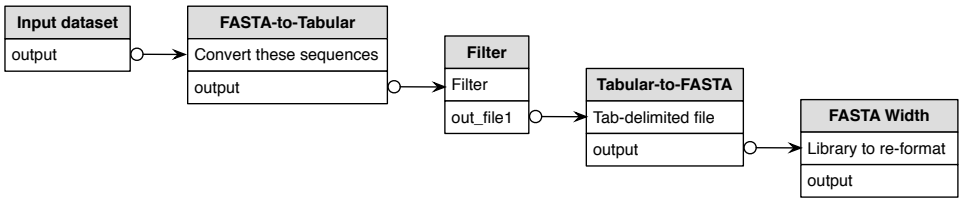


Figure 1.4: Galaxy genomics workflow (*myExperiment workflow 3030*).

linear workflow which was created in Galaxy and then published to myExperiment[6]. Example 1.2.4 is a general purpose data cleaning workflow for pre-processing textual data. It is non-linear and features a nested workflow – a separate sub-workflow invoked from within the top-level workflow[7]. Example 1.2.5 is a highly system-specific RapidMiner machine learning workflow that is visually more detailed than the previous examples[8]. The diagrams have been reformatted for presentation here, but without loss of semantics. Links to the originals are included in each case.

Example 1.2.3 (Galaxy genomics workflow) *A single linear sequence of components, each feeding into the next like a waterfall process; Galaxy workflows do not support loops. The workflow diagram, Figure 1.4, is augmented by a corresponding set of parameters to each step that are not depicted visually but do accompany the workflow. The full description of the "Transform 'Stitch Gene blocks' FASTA blocks to standardized FASTA file" workflow is reproduced below.*

```
Annotation:  Converts FASTA blocks to a FASTA file.
Step 1:  Input dataset
Output of tool "Stitch Gene blocks":  select at runtime
Step 2:  FASTA-to-Tabular
Convert these sequences:  Output dataset 'output' from step 1
How many columns to divide title string into?:  1
How many title characters to keep?:  0
Step 3:  Filter
Filter:  Output dataset 'output' from step 2
With following condition:  len( c2.replace( '-', '' ) ) > 0
Number of header lines to skip:  0
Step 4:  Tabular-to-FASTA
Tab-delimited file:  Output dataset 'out_file1' from step 3
Title column(s):  1 (value not yet validated)
Sequence column:  2 (value not yet validated)
Step 5:  FASTA Width
Library to re-format:  Output dataset 'output' from step 4
New width for nucleotides strings:  50
```

Source: myExperiment workflow 3030.

Example 1.2.4 (Taverna text processing workflow) *A non-linear workflow to convert text to lowercase and remove non-alphanumeric characters and stopwords (commonly occurring words in a language). The workflow diagram, Figure 1.5, exposes details of web service parameters and workflow inputs, as well as intermediate variable names. Data types are hinted at in variable names but not explicitly depicted.*

[6] `http://www.myexperiment.org/workflows/3030.html`, visited April 2014.
[7] `http://www.myexperiment.org/workflows/1750.html`, visited April 2014.
[8] `http://www.myexperiment.org/workflows/3208.html`, visited April 2014.

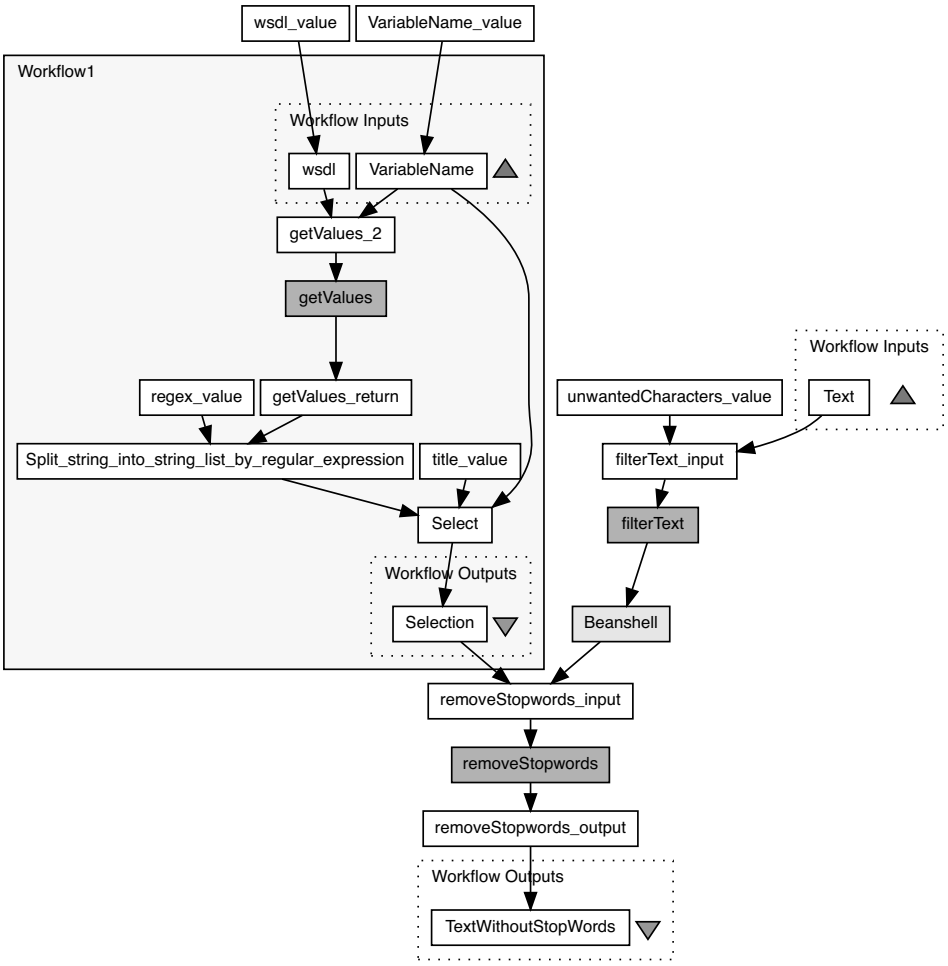


Figure 1.5: Taverna workflow (*myExperiment workflow 1750*).

An embedded sub-flow, labelled 'Workflow 1' is shown. The workflow diagram is documented by a natural language description which we reproduce below.

> The input to this workflow is plain text. The text is preprocessed so that non-alphanumeric symbols are removed, the text is transformed to to lower case and stopwords are removed. The workflow first removes the characters from this set: `~!@#'$%&*()_+=-\{}|][":;'?><,./.` Then it transforms the text to lower case. The user will be prompted to select a dictionary for stopwords from a list. The workflow will, based on the elected list, remove the stopwords. Stopwords are words that do not carry meaning, like, the, an,... The web service for stopwords removal integrates six English stopwords dictionaries and one for the Slovenian language. The output of the workflow is text in lower case without non-alphanumeric characters and without stopwords.
> *Source: myExperiment workflow 1750.*

Example 1.2.5 (RapidMiner machine learning workflow) *A multiple-output workflow for choosing the best k value for the k-Nearest Neighbour classification of the Breast Cancer Wisconsin (Diagnostic) data set. The format of the workflow diagram, Figure 1.6, is completely specific to RapidMiner and requires the reader to learn*

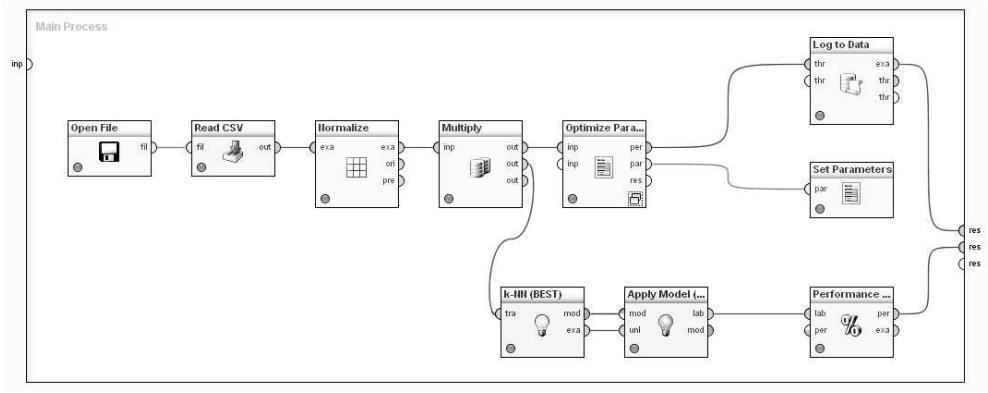


Figure 1.6: RapidMiner workflow (*myExperiment workflow 3030*).

its notation in order to completely understand its function. An informative natural language description accompanies the workflow and is reproduced below.

> `The process determines the best value for the parameter k for the k-NN classification of the Breast Cancer Wisconsin (Diagnostic) data set available in the UCI Machine Learning Repository. The optimal k is computed by using 10-fold cross-validation. (To get better results each cross-validation is repeated 10 times and the averages of the runs are considered.) Finally, a k-NN classifier is built and evaluated on the entire data set using the optimal k. During the process the resulting average performances are logged for each k.`
> *Source: myExperiment workflow 3208.*

The workflow diagrams featured in the remainder of this Thesis are visually inspired by Taverna workflow diagrams, of the style of Figure 1.5, but are not specific to any particular workflow enactment system. By avoiding system-specific details and remaining at a high level of abstraction our workflows are both more easily understood and implemented for any given enactment system. We formally define an *abstract workflow diagram* as follows.

Definition 1.2.6 (Abstract Workflow Diagram) Abstract Workflow Diagram *is a schematic representation of a workflow where rectangular vertices represent activities enacted by web services, double rectangular vertices represent activities enacted asynchronously by background processes, rounded vertices represent presentation-layer data transformation activities, and edges are labelled with (informal) data types. Activities enacted in web services correspond to one or more remote API method calls. Input ports are represented as downward pointing triangles and output ports as upward pointing triangles. Data is delivered to input ports and received from output ports.*

The illustrative example in Figure 1.7 is an abstract workflow diagram representing a workflow with two input ports and three output ports. All ports and components, services or generators are labelled with example names but would normally have more meaningful names. Data arriving at *Input 1* is transformed from data of *Type 1* by *SERVICE 1* to data of *Type 3*; other flows can be read similarly. Some concrete data types in this Thesis are CSV, HTML, JSON, Text, XML and Prolog terms. Although data types in this diagram are all assigned different names, they

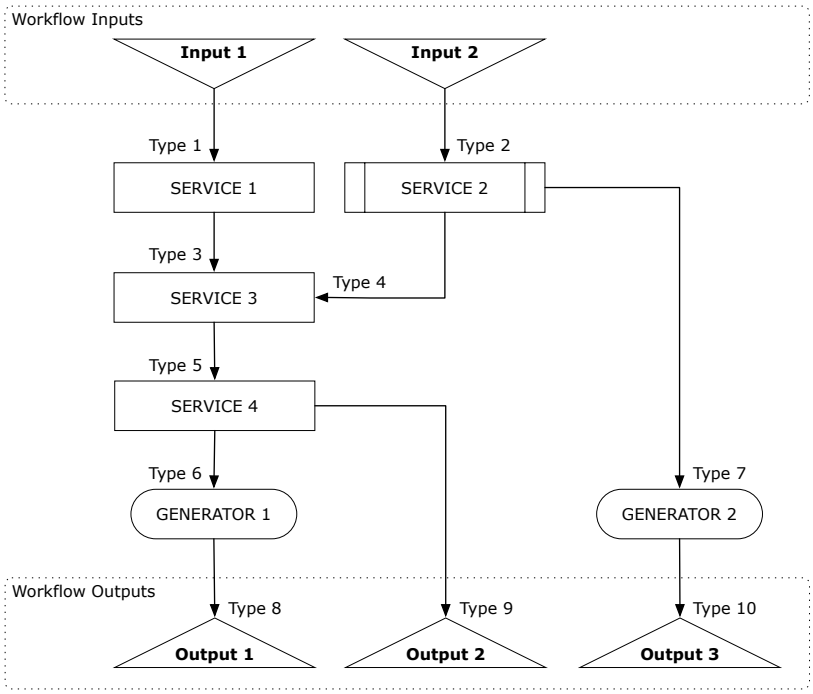


Figure 1.7: Illustrative example of an *abstract workflow diagram*.

need not be different. All the components here are synchronous, returning their results immediately, apart from *SERVICE 2* which is asynchronous, meaning that calls to its methods are non-blocking, returning immediately and leaving the service to work in the background. Thus *SERVICE 3* and its dependents cannot complete until *SERVICE 2* completes. Examples of asynchronous services in this Thesis are a web crawler and long-running transformations of large datasets. An abstract workflow diagram does not specify further details of the implementation of either the web services or the overall enactment of the workflow.

Interestingly, in terms of curating such research workflows over many years and their value to future researchers, the diagrammatic representation may well have a longer useful life than the executable workflow itself because the former has fewer dependencies [LRL+12]. Without continuous use or active curation, software and systems tend to become unrunnable over time. This phenomenon has already been observed for e-Science workflows [BRGC+12]. It should be noted that this degradation is not unique to workflows and applies to all research software. However, even faced with this potential loss of functionality, workflow diagrams, and workflows in general, have another important role in recording the provenance of research data and results. Recent work on *research objects* and *computational research objects* seeks to address the long-term archiving and curation of research data and workflows [BCG+12, DR13]. Although research data management is broadly outside the scope of this Thesis, the models that emerge in our work do have benefits in this area and will be touched upon briefly in "Chapter 7 - Discussion".

Web Services

Workflow systems typically support the scheduling and monitoring of locally running component processes, for example running a shell script on the user's own workstation or on a cluster or a supercomputer that the user has access to. While the advantages of executing components in this way are manyfold, particularly in the area of performance and minimising data transfer overheads, there are some significant disadvantages in the context of integrating workflows into third-party systems such as web applications. Running a workflow as a third-party in this way requires the installation of the software and all the dependent libraries as well as access to the necessary hardware. Avoiding the need to replicate and maintain functionality locally for every client application is the primary motivation for the services model.

When considering the choice of a suitable protocol for invoking remote services there are plenty of choices, including remote procedure calling via RMI or CORBA, message queues, or the highly abstracted OSGi framework. At this point an important distinction must be made between invoking components as services and the interaction of components on the same physical system. With the former, there is an upload/download cost to getting data into/out of the system; the same is not necessarily true of the latter in cases where there is a shared memory model (e.g. a common local database or file system). In the context of the use cases in this Thesis, it was clear from the outset that working data could be shared amongst the component services, thereby restricting the need to transfer large amounts of data to just upload at the start and download at the end. We discuss this issue further in Chapter **??** but for the purposes of this introduction it is suffice to note that invocation of components does not (normally) require the transfer of large amounts of data. For this reason, the software frameworks in this Thesis use web services to invoke components in a workflow whether local or remote – the engineering advantages of such loose coupling of system components outweight other costs in our setting. Our use cases centre on web applications and so we are predominantly interested in *web services*.

Even though it is less relevant to our investigations, we mention Grid computing because of its co-evolution with the field of e-Science that motivated our own workflows. Initially, e-Science was closely tied to a set of distributed resource sharing protocols, named Grid after the concept of electrical power distribution grids. As the field has matured, there has been a shift away from Grid architectures towards applications that operate over simple web service protocols [WHW10]. Although still relevent to digital science, e-Science no longer emphasises Grid and neither does our own work. Whilst noting that anywhere in this Thesis that services are mentioned Grid equivalents could be substituted, we will not refer to Grid again.

The idea of web services is simple: allow remote access to distributed, shared computational resources and data via an application programming interface (API) accessed through established Internet protocols. Apart from the advantage of prospective users being able to use software without having to install it, the web services model has a number of other benefits, including the ability for users to: make their own application-specific customisations using the services with workflow tools; select other data sources and, importantly, integrate the software with their own research tools without having to modify code. Web services come in two flavours: SOAP and REST.

SOAP – originally *Simple Object Access Protocol*, exchanges structured data as XML over some transport protocol – most typically one of HTTP, JMS, SMTP or TCP [soa07]. SOAP services are typically described by a formal specification in WSDL (Web Service Description Language) of their parameters and associated data types, superficially making these services processable by tools. In practice, SOAP is fraught with versioning problems across heterogeneous systems and the underlying XML representation is unsuitable for the efficient transfer of binary data without introducing further protocols. Despite this, it is widely supported by general purpose software development tools.

REST – *Representational State Transfer Protocol*, is an easily understood design pattern for web services based around the ubiquitous HTTP protocol and its familiar vocabulary of URIs, media types, requests and responses [FT02]. Although easily understood by humans, REST offers little in the way of guidance concerning the machine-readable description of parameters and data types, which inevitably makes their automated support in tools rather basic – requiring the developer to supply most of the details. However, REST does not require the use of XML and *RESTful* services commonly use JSON, CSV, plain text and binary data. Versioning problems exist in REST but these are the responsibility of individual services rather than inherent in the protocol itself.

The choice of RESTful web services over SOAP-based web services throughout this Thesis was not straight forward. Popular workflow tools, such as myExperiment, Kepler, RapidMiner and Taverna all support REST and SOAP to varying degrees. However, SOAP is well established in e-Science, as evidenced by the BioCatalogue[9] life sciences web services directory which, at the time of writing, has 2210 SOAP services as compared to only 146 REST services (April 2014). Even so, the compelling argument in favour of REST for the software frameworks described in this Thesis is that lightweight RESTful services are far more easily integrated into the Web 2.0 applications in our use cases. SOAP's reliance on XML considerably increases the complexity and decreases the efficiency of browser-based JavaScript code, where JSON is the natural serialisation. For developers in general there is a growing trend away from SOAP-based services towards the simpler RESTful ones, for all the reasons espoused in [EHT10], and this was also an important factor in our choice.

1.2.3 Dataspaces

The notion of a *dataspace* [FHM05, HFM06, FHM08] accurately captures the ideas behind our work in "Chapter 5 – Querying and Merging Heterogeneous Data" where we are concerned with database-like queries against data originating from multiple sources with different representations, structures and conventions. In the context of the research intelligence domain, the data matching aspects of the web applications described in our use cases can be phrased in terms of queries against such data.

Here we introduce dataspaces by first describing some of the problems that they solve in the closely related field of *data integration*. Data integration aims to bring together data from multiple sources, as just mentioned, and combining it to create a

[9] BioCatalogue – `http://www.biocatalogue.org/services`, visited April 2014.

unified view of the data, as if it were a single well-designed database. Having this unified view of data makes it possible for users to query the data, generate, reports and (in some few cases) update the data just as they would had the data originated from a single database. Whether integration is performed manually or automatically, it is not always an error free process and there is often a degree uncertainty in the end result. While inaccuracies and noise in data inevitably give rise to uncertainty, simply ensuring complete and accurate data does not guarantee certainty in its integration. This unavoidable uncertainty arises from a central problem in data integration: the difference between *syntactic integration* and *semantic integration*.

Syntactic integration connects information from diverse data sources by performing exact, or sometimes with approximate matching to detect typos or minor variations in formatting. The implicit inference is that a syntactic match implies a semantic match. A semantic match means that the two data being compared refer to the same entity. This, of course, is often not the case. For example, an approximate match of the word “apple" in one data source may refer to the company Apple Inc. whereas in another it may refer to the fruit of the same name. Background or contextual knowledge is required in order to strengthen a syntactic connection up to a full semantic connection and in order to estimate a degree of confidence in that connection. The reason why this presents such a problem in data integration is not that semantic integration cannot be achieved through a combination of automatated and manual mappings between different sources of data but rather that this process is time consuming and expensive. We discuss this issue further in “Section 6.2.1 – Schema and Ontology Matching” (p.185). To integrate data efficiently requires knowledge of the kinds of queries that will be made against the unified view so that appropropriate semantics can be applied to the mappings between the data.

By contrast, the dataspaces approach to data integration is to offer a base-level of querying data from different sources but to only implement additional integration as and when the need arises. For example, in a DataSpace Support Platform (DSSP) [SDH09, SDH11] base-level querying consists of keyword search across all data in a dataspace; when further integration requirements arise, for example database-like queries or data mining, then further effort can be invested through the use of semi-automated matching tools to create the necessary semantic integration. Informally, this approach is often described as ‘pay as you go’ data integration. Data sources can be extremely broad and varied, for example including desktop search as well as document repository search in addition to the more traditional data integration domain of relational databases. In Section 2.2.1 (p.30) we describe the data sources[10] that are most relevant to this Thesis, all of which one might expect to find in a dataspace. The components of a dataspace, as proposed in [FHM05], are:

- Dataspace Services – such as search and query.
- Dataspace Systems – local store and indexing of data sources.
- Catalogue – an inventory of data elements and relationships.

Searching and querying services also extend to allow searching of metadata (i.e. data describing the type of and relationships between data) about the various data sources.

[10] A formal definition of a data source also appears in the same chapter, as Definition 2.1.1 (p.27).

A dataspace will also normally be expected to offer browsing of data and metadata, data discovery (e.g. searching a file system for documents) and monitoring of the participating data sources (e.g. to detect changes in metadata or the data itself).

Later, in "Chapter 5 – Querying and Merging Heterogeneous Data", we introduce a formal framework that from the perspective of dataspaces, would be described as a dataspace service. The objectives of that framework have much in common with the objectives of dataspaces.

In this section we introduced the software architectures and theoretical foundations that forms the basis of the frameworks developed in the later chapters of this Thesis: dataflows (Chapter 4), workflows (Chapter 3) and dataspaces (Chapter 5). In the next section we discuss the research questions and objectives for our work.

1.3 Research Questions and Objectives

This Thesis explores software and theoretical frameworks for building tools to profile and match heterogeneous data about researchers and their organisations. The work predominantly addresses software engineering research questions that arise in this research intelligence setting, particularly where the tools are implemented as interactive web applications. In doing this we are interested in combining research from the separate traditions of higher-order computational logic and workflow systems.

Our intuition is that the applications described in our motivating use cases share many common functional aspects and can be viewed as specific instances of some generic submission sifting workflow composed from a family of re-usable web service components for profiling and matching heterogeneous data. Moreover, we anticipate that general frameworks designed around these concepts can substantially simplify engineering the web applications – reducing the development time and the number of lines of code required to implement these web applications by an order of magnitude as compared to an implementation without using a framework.

It needs to be stressed that the focus of this work is not on advancing knowledge in the underlying profiling and matching algorithms *per se*; the focus is more on making it easier to engineer web applications that incorporate existing algorithms. Indeed, one of our initial research questions is whether any established, well-studied profiling and matching algorithms can be given novel utility through implementation as web services and workflows? A converse question is whether there are characteristics of web services and workflows that facilitate the use of existing algorithms, for example by exploiting data structure and data type metadata. In both cases, the algorithms are invariant; the questions concern their packaging and use.

All our use cases hinge on successfully answering the question: can the submission sifting use case be decomposed into separately re-usable components of a generically useful submission sifting workflow? If so, which reconfigurations of that workflow and its components are required to satisfy each of our use cases? What are the advantages and disadvantages of this approach for web application development compared to purpose-written implementations? Can applications produced in this way be shown to be deployable and useful in real-world settings? What are the

limitations of a generic submission sifting workflow?

There are a number of research questions concerning the data involved in our use cases. Some of these questions relate to the choice of input data itself and others relate to what is rather unscientifically often referred to as 'data cleaning'. We observe that 'data cleaning' is actually data transformation and as such is an important part of an overall workflow that should be transparent, clearly documented and reproducible in accordance with the scientific method. Therefore we ask whether such data transformations involved in addressing our use cases can be easily made transparent using a workflows and components for their implementation? Our use cases afford an opportunity to explore this question in detail by introducing various levels of data transformation to the process of acquiring reviewer publication information from online bibliographies or web pages. Examples of such pre-processing include: extraction of document metadata such of titles, abstracts, authors and date; removal of HTML mark-up; removal of commonly occurring words; and, constraining the vocabulary to words that occur in a domain-specific lexicon.

Related research questions arise from the possibility that these data transformations will depend heavily on the data sources chosen for a particular web application – for example, whether the same online bibliography must be used for all researchers being profiled and matched, or whether researcher homepages or even full-text of their publications is used or whether some mix of these is practical? For some of the use cases, should the profiles of individual researchers be compared one against the rest (i.e. where all the other researchers are considered as one single aggregate individual) versus pairwise Cartesian comparison of each researcher to each other?

Intuitively, adaptability and extensibility are two practical engineering advantages of the workflow model. Our use cases provide an opportunity to investigate these questions here too. For instance, how easy is it to take into consideration the gradual drift of a researcher's interests over time due to advances in the field's frontier or due to shifts in the researcher's own personal focus? Can this temporal dimension be incorporated into workflows by pre-processing the data or selecting the data to only include publications from the last n years when calculating reviewer profiles? In a wider sense, how easy is it for web application developers to incorporate their own data pre-processing, profiling and matching algorithms into the workflows? Another instance of the need for adaptability and extensibility arises where researcher homepages are used as input to the computation of profiles. Here the web application designer has to decide whether to include hyperlinked pages branching off from a researcher's homepage to other web pages on the same website, or even to external sites. Such hyperlinks often link to sub-pages of the homepage, which may contain further information about the researcher and their interests, their research group, their projects and so on. They may also link to external sites related to their interests. For any given application, decisions must be made as to which classes of links to follow (e.g. all; same domain; external; ones not shared with other researchers) and whether to recursively follow links to some cut-off depth. Can this adaptability and extensibility be facilitated by the approach and without loss of transparency of this data pre-processing?

Implicit in all the use cases is that heterogeneous data may be extracted, transformed and loaded (i.e. the Extract, Transform, Load (ETL) stage of business intelligence) from diverse sources such as web pages, digital libraries, knowledge bases,

the Semantic Web and databases. Much of the heterogeneity of the data involved in our use cases arises in its structure rather than its underlying textual type. The ease with which the web application developer can manipulate that structure and the ease of incorporating textual pre-processing have already been touched upon. However, non-textual aspects of the data do need to be considered in certain use cases. Taking the relative dates of research publications is one example we have already mentioned above. Background knowledge, such as *conflict of interest* boolean relations between reviewers, inject other data types. Prior knowledge about the relative importance of specific words or phrases could be expressed as term-weight (string-real) pairs and require incorporation into the similarity calculation. Other examples arise when combining comparisons of multiple data sources or types. For instance, in the submission sifting use case, if the programme committee chair chooses to finesse the calculation of similarity between the reviewer profiles and paper profiles by adjusting the relative contributions of keywords, titles and abstracts to the overall similarity calculation, then the features being compared might be similarity scores.

Finally, serendipitously, are there deeper design patterns or wider applications suggested by investigating frameworks for research intelligence applications? If so, can we describe them and perhaps make initial steps towards their investigation?

We distil these research questions down to arrive at the following four summary research objectives for this Thesis.

O1. Decomposition of the submission sifting use case into separately re-usable components of a flexible framework for profiling and matching textual content – capable of implementing our research intelligence use cases.

O2. Description and implementation of a general purpose framework for profiling and matching heterogeneous data in a web application context.

O3. Identification and explorative investigation of deeper design patterns or wider applications suggested in addressing objectives **O1-2**.

O4. Demonstration of the joint potential for *higher-order computational logic* and *workflow systems* in addressing software engineering problems.

1.4 Contributions

As a result of addressing the above research objectives and the research questions they entail, this Thesis contributes to knowledge in the following four ways.

C1. Exploration of workflow approaches to the *submission sifting* problem
Submission sifting is the problem of matching submitted conference/journal papers to potential peer reviewers based on the similarity between the paper's abstract and the reviewer's publications as found in online bibliographic databases. A generic submission sifting workflow and its components are abstracted to define a general framework to enable web services to be assembled into workflows to analyse heterogeneous textual content from web pages and documents. A range of workflows are introduced to investigate the utility of

this framework in creating applications to support scientists and their organisations. These applications are, of themselves, not individually novel; the novelty is in their collective definition as workflow compositions of web services. SubSift implementations of these workflows constitute a proof of concept realisation of this general profiling and matching framework. One of the demonstrator applications is currently used as a recommender system by editors of two leading computer science journals[11]. To the best of our knowledge, despite the potential utility of such a set of services and notwithstanding that the underlying techniques are well established, such an engineering solution is not immediately available elsewhere.

C2. Dataflow model for analysing heterogeneous data

A higher-order dataflow model is introduced. The model ranges over a class of higher-order relations that are sufficiently expressive to represent a wide variety of unstructured, semi-structured and structured data. A proof of concept web services implementation of higher-order dataflows, JSONMatch, is used to demonstrate that the combination of this model and higher-order representation provides a powerful framework for analysing heterogeneous data in a web application context. It is shown that the model has the necessary expressive power to be able to implement submission sifting workflows through concise user-supplied higher-order parameters to a dataflow at runtime rather than through extensive application-specific code that would otherwise be required. It was shown that pure transformations in the model satisfy the conditions of an *embarrassingly parallel* function and hence, in principle at least, are highly parallelisable and highly scalable.

C3. Formalism for querying heterogeneous structured data

The relational model and its associated relational algebra provide a formalism for querying relational data in SQL databases. Hitherto there has been no closely corresponding model and algebra for querying structured data originating from heterogeneous sources. A higher-order relational model is introduced, lifting the traditional relational representation to the basic terms in a higher-order logic that is better suited to the representation of structured data. A relational algebra for basic terms is defined as a single coherent formalism for querying heterogeneous structured data. It is shown that the traditional relational algebra is a special case of the introduced algebra. An extension incorporates approximate joins on complex structured data and is demonstrated to be both feasible have promise for future work.

C4. Bridge between two separate fields of research

Our work brings together the otherwise disparate research traditions of higher-order computational logic and workflow systems; as such, this Thesis constitutes a bridge between these communities and serves to demonstrate their joint potential for addressing problems in research intelligence.

[11] Reported by personal communication by Flach (January 2010), Editor-in-Chief, *Machine Learning*, and Webb (January 2013), Editor-in-Chief, *Data Mining and Knowledge Discovery*.

1.5 Publications

The core content of this Thesis has been peer-reviewed and published in two journal papers and three conference papers, listed below. My contribution in each case has been to design the methodology, develop the models and demonstrations, perform experiments and critical analysis, and write the paper – all under the supervision of Flach. Co-authorships acknowledge the work of Spiegler, whose implementation of the vector space model was used in the motivating use case prior to my development of SubSift, and of Bailey and Rogers who provided valuable user feedback.

- **Simon Price** and Peter A. Flach. A higher-order data flow model for heterogeneous Big Data. In *IEEE International Conference on Big Data, Santa Clara, USA, October 2013, Proceedings*. IEEE Computer Society, 2013.
- **Simon Price**, Peter A. Flach, Sebastian Spiegler, Christopher Bailey, and Nikki Rogers. SubSift web services and workflows for profiling and comparing scientists and their published works. *Future Generation Computing Systems*, 29(2):569–581, 2013.
- **Simon Price**, Peter A. Flach, Sebastian Spiegler, Christopher Bailey, and Nikki Rogers. SubSift web services and workflows for profiling and comparing scientists and their published works. In *IEEE Sixth International Conference on e-Science, Brisbane, Australia, December 2010, Proceedings*, pages 182–189. IEEE Computer Society, 2010.
- **Simon Price**, Peter A. Flach, and Sebastian Spiegler. SubSift: a novel application of the vector space model to support the academic research process. *Journal of Machine Learning Research - Proceedings Track*, 11:20–27, 2010.
- **Simon Price** and Peter A. Flach. Querying and merging heterogeneous data by approximate joins on higher-order terms. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 2008, Proceedings*, volume 5194 of *LNCS/LNAI*, pages 226–243. Springer-Verlag Berlin Heidelberg, 2008.

The Thesis is also supported by seven supplementary publications, five of which were peer-reviewed, the other two being technical reports. These are listed below along with the chapter each supports. Unless otherwise specified, my contribution has been to perform literature search, design the methodology, develop the software, perform experiments and critical analysis, and write the paper.

- **Simon Price** and Peter A. Flach. Mining and mapping the research landscape. In *Digital Research 2013 Conference*. University of Oxford, September 2013.
  Supports "Chapter 3 - Workflows for Profiling and Matching Textual Content".
- **Simon Price** and Peter A. Flach. A relational algebra for basic terms in a higher-order logic. *Technical Report CSTR-13-004*, Department of Computer Science, University of Bristol, July 2013.
  Supports "Chapter 5 - Querying and Merging Heterogeneous Data".

- Michael A. Covington, Roberto Bagnara, Richard A. O'Keefe, Jan Wielemaker and **Simon Price**. Coding guidelines for Prolog. *TPLP*, 12(6):889–927, 2012.
  Co-wrote this *de facto* best practice **journal paper** with renowned Prolog textbook authors and engineers. My contribution arose from and guided code in "Chapter 5 - Querying and Merging Heterogeneous Data".

- **Simon Price**, Peter A. Flach, and Sebastian Spiegler. SubSift: a novel application of the vector space model to support the academic peer review process. In *Workshop on Applications of Pattern Analysis (WAPA 2010)*. Windsor, UK, September 2010.
  Supports "Chapter 3 - Workflows for Profiling and Matching Textual Content".

- Peter A. Flach, Sebastian Spiegler, Bruno Golénia, **Simon Price**, John Guiver Ralf Herbrich, Thore Graepel, and Mohammed J. Zaki. Novel tools to streamline the conference review process: Experiences from SIGKDD'09. *ACM SIGKDD Explorations*, 11(2):63–67, December 2009.
  Supports "Chapter 3 - Workflows for Profiling and Matching Textual Content". Contributed work-in-progress text and code.

- **Simon Price**. A review of the state of the art of machine learning on the Semantic Web. In *Proceedings of 2003 UK Workshop on Computational Intelligence*, pages 292–299. University of Bristol, September 2003. Extended version published as: *Technical report CSTR-04-005*, Department of Computer Science, University of Bristol, October 2004.
  Supports "Chapter 6 - Related Work".

1.6 Structure of Thesis

In this first chapter we have introduced our motivational use cases, defined the scope of the theoretical and software frameworks relevant to this work, set out our research questions and objectives, highlighted our contribution and listed a range of supporting publications. The remainder of this Thesis is organised as follows.

Chapter 2 - Background: first reviews a variety of schemes for representing and accessing heterogeneous data before moving on to review similarity measures for comparing heterogeneous structured data.

Chapter 3 - Workflows for Profiling and Matching Textual Content: explores different solutions to the submission sifting problem and describes how an workflow-based approach resulted in the SubSift framework for comparing general textual content, which also turned out to be highly applicable to our other use cases.

Chapter 4 - Higher-Order Dataflows: generalises the SubSift framework to define a higher-order dataflow model which we show to be embarrassingly parallel. We compare the model with SubSift using JSONMatch, our proof of concept implementation.

Chapter 5 - Querying and Merging Heterogeneous Data: speculates that the data comparisons in our use cases may be expressed as queries in a relational algebra defined on terms in a higher-order logic, and reports on initial results.

Chapter 6 - Related Work: surveys existing systems that cover parts of the problem of comparing heterogeneous data and contrast these with our own work. Prior work relating to our motivational use cases is also reviewed and related to our approach from an engineering perspective.

Chapter 7 - Discussion: examines the limits of our work and reports on key lessons learnt from developing frameworks to address our use cases.

Chapter 8 - Conclusions and Future Work: rounds up both the planned and serendipitous results of our exploration and then discusses a number of possible future directions for both the approach and specific implementations featured.

Chapter 2

Background

This chapter begins with an introduction to the terminology used in this Thesis and is relevant to all readers. The remainder of the chapter is divided in two: the first part surveys *informative background topics* relevant to our work but about which readers only require a general awareness; the second introduces the *required background theory* necessary to read and understand this Thesis.

2.1 Terminology

We first introduce some notation relating to the concept of a *data source* and then define *heterogeneous data* so that we may introduce the notions of profiling and matching approaches independently of specific data representational formalisms. To do this we consider a data source to be a collection of descriptions of members of a set. Our definition, in part, borrows from notation used in the *record linkage* literature [FS69] which we discuss later in "Chapter 6 – Related Work".

Definition 2.1.1 (Data Source) *For a set P, a data source Δ_P is a multiset whose elements are descriptions $\delta(x)$ of the elements $x \in P$.*

Recall that a multiset is a generalisation of a set whereby repeated elements are considered. A multiset reduces to a set if each element occurs zero or once only.

As an example of a data source, let P be the set of people $x \in P$ working at the University of Bristol and the data source Δ_{people} be their contact details. Each description $\delta(x)$ in Δ_{people} is the name, email and address of a person $x \in P$. So, in a relational representation, $\delta(x)$ might be an abbreviation for a record consisting of the set of attribute-value pairs associated with x such that,

$$\delta(x) = \{(\text{name}, name_x), (\text{email}, email_x), (\text{address}, address_x)\}.$$

Clearly, when representing the elements x of a set in some data source it is necessary to choose a description function δ. This choice is dependent on the application domain and problems being addressed but in practical terms amounts to a choice of a data representational formalism. Later in this chapter we review some commonly used data representations but here we restrict our attention to the following abstract characteristics of variations in a description $\delta(x)$.

- Descriptions may be *syntactically* different, e.g. using `full_name` instead of `name` in the above example.
- Descriptions may be *structurally* different, e.g. using `first_name` and `last_name` instead of `name` in the above example.
- Descriptions may be *semantically* different, e.g. using `address` to refer to a home address instead of a university office address.

These differences can also extend to (or sometimes only occur in) the data values themselves, for instance $name_x$ = "Price, S." might simply be a syntactic variation of $name_x$ = "Simon Price". In later chapters we give more detailed examples of variations between descriptions but for now we just note that such differences exist.

We use the term *heterogeneous data* to collectively refer to sets of data sources with different description functions δ as defined below.

Definition 2.1.2 (Heterogeneous Data) *A set of data sources* $\{\Delta_{\alpha_1}, \ldots, \Delta_{\alpha_n}\}$ *with respective description functions* $\delta_{\alpha_1}, \ldots, \delta_{\alpha_n}$ *and* $n \geqslant 2$, *for some sets* $\alpha_1, \ldots, \alpha_n$, *is* heterogenous data *if* $\delta_{\alpha_i} \neq \delta_{\alpha_j}$ *for all* $i, j \in \{1, \ldots, n\}$.

Although only occassionally mentioned in this Thesis, *homogeneous data* refers to sets of data sources that are not heterogenous data.

Definition 2.1.3 (Homogeneous Data) *A set of data sources* $\{\Delta_{\alpha_1}, \ldots, \Delta_{\alpha_n}\}$ *with the same description function* δ *and where* $n \geqslant 2$, *for some sets* $\alpha_1, \ldots, \alpha_n$, *is* homogenous data.

The terms homogenous data and heterogenous data refer to differences between description functions δ_α in a set of data sources. This is one way in which data sources can differ; another is in the nature of the data values in the co-domain of description functions, i.e. the data values in the data source. Where such values are indivisible "atoms" they are referred to as *atomic data*, e.g. integers, reals, booleans, characters. By contrast, where such values consist of more than one atomic data values compounded together by grouping and/or ordering they are referred to *structured data*, e.g. sets, multisets, lists, trees, graphs.

Definition 2.1.4 (Structured Data) *Any element* y *in the co-domain of a description function* δ *is referred to as* structured data *iff* y *is not an atom.*

Whether data is heterogenous, homogenous or structured, it is important to note that equality of a pair of descriptions in a data source does not guarantee that they refer to the same underlying set member. For instance, two people with identical names may appear in a list of staff in a university department or as authors in an online bibliography. Conversely, inequality of a pair of descriptions does not guarantee that the descriptions do not refer to the same underlying set member. For instance, a person may appear in the contacts data source more than once because they have more than one position at the university or because of variations in spelling or typos. So, instead of testing a pair of descriptions for an equality relation, matching must test for an equivalence relation between profiles.

Definition 2.1.5 (Profiling) Profiling *is a function* ∇ *with signature* $\nabla : \Delta_P \rightarrow \Delta'_P$, *where* Δ_P *and* Δ'_P *are data sources with elements* $\delta(x)$ *and* $\delta'(x)$ *respectively for some* $x \in P$.

A description $\delta'(x)$ is called the *profile* of $\delta(x)$. The definition of profiling does not require that a profile $\delta(x)$ discriminates x from all the other elements in P, but the more discriminating a profile is the more useful it is. For instance, the email address of a person is far more discriminating as a profile than a person's institution name.

As natural language shorthand, we often informally treat $\delta(x)$ as a proxy for x, effectively saying that $\nabla(\delta(x))$ is $\nabla(x)$, the profile of x (which it is clearly not; it is the profile of $\delta(x)$). For example, we treat the published works of researchers in this way and sometimes informally write that the "profile of the researcher's publications" is the "profile of the researcher".

We have already informally used the term *matching* in our introductory chapter. We now more formally define *matching* based on the above concepts of profiles and data sources.

Definition 2.1.6 (Matching) *Let* Δ_P, Δ'_P *and* Δ''_P *be data sources with elements* $\delta(x)$, $\delta'(y)$ *and* $\delta''(z)$ *respectively for some* $x, y, z \in P$. Matching *is a function* $\approx$ *with signature* $\approx : \Delta_P \times \Delta'_P \rightarrow \Delta''_P$, *such that*

$$\Delta_P \approx \Delta'_P \;=\; \{\, \delta''(z) \mid \delta(x) \sim \delta''(z) \wedge \delta'(y) \sim \delta''(z),\ \delta(x) \in \Delta_P, \delta'(y) \in \Delta'_P \},$$

where $\sim$ *is a predicate with signature* $\sim : \delta(x) \times \delta'(y) \rightarrow \Omega$, *where* Ω *is the type of the booleans, such that* $\delta(x) \sim \delta'(y) \;\Rightarrow\; x = y$.

Informally, $\sim$ is an approximate equality predicate and Δ''_P is a set of descriptions $\delta''(z)$ of approximately equal pairs from $\Delta_P \times \Delta'_P$. The definition of $\sim$ does not require that a $\delta(x) \sim \delta'(y) = \top$ is only true when $x = y$, i.e. that there is not some other $y \in P$ for which $\delta(x) \sim \delta'(y) = \top$. However the fewer y for which this is true, the more useful the $\sim$ function is for matching. In an ideal matching function $\sim$ will satisfy $\delta(x) \sim \delta'(y) \Leftrightarrow x = y$, only matching when the descriptions represent the same element in P.

In this section we introduced our terminology for *data sources*, *heterogeneous data*, *homogenous data*, *structured data*, *profiling* and *matching*. These terms are widely used throughout the Thesis. Over the remainder of this and subsequent chapters, further terms are defined at the point at which they are first used.

2.2 Informative Background Topics

In this section we survey background topics which informed our Thesis. Only a general awareness of these topics is required in order to understand the Thesis and some readers may wish to skim these on a first reading. The material begins with a description of structured data and then examines the three most commonly used structured data representations: relational, document-based and graph-based. We

then survey three foundational topics relating to the problem of comparing heterogeneous data, which have some bearing on our profiling and matching tasks: database deduplication, record linkage and information retrieval. The section then concludes by reviewing techniques for comparing structured data. The topics covered listed below.

- Representing Structured Data
  - Relational Representations
  - Document-based Representations
  - Graph-based Representations
- Comparing Heterogeneous Data
  - Database Deduplication
  - Record Linkage
  - Information Retrieval
- Comparing Structured Data
  - Similarity and Distance
  - Distances for Structured Data

Each section makes reference to where in the Thesis these topics become most relevant. Similarly, later chapters refer back to topics in this section from various points, as do later sections in the current chapter.

2.2.1 Representing Structured Data

The data we are interested in profiling and matching in this Thesis is *structured data* (Definition 2.1.4) and so the representation of such data is an important consideration. Structured data includes lists, sets, multisets, trees and graphs, and the three representational formalisms covered in this section have their respective strengths and weaknesses for each of these structures. When reading the literature and our survey below, it should be noted that structured data also includes a range of commonly used representations known as *semi-structured data* (e.g. HTML and XML) and, so-called, *unstructured data* (e.g. text). Semi-structured data is textual data containing embedded 'tags' that associate areas of the text with some pre-defined schema to designate, for example, titles, headings, paragraphs and hyperlinks. Unstructured data is typically textual data that does not have a pre-defined schema but which may contain names of people, places and events as well numbers, ratios, dates and units.

In "Chapter 3 – Workflows for Profiling and Matching Textual Content" we are primarily concerned with profiling and matching unstructured data, so much so that other types of data (e.g. HTML) are converted into unstructured data beforehand. From Chapters 4 onwards, the scope of the Thesis expands to include all structured data. The data representation used in these later chapters is described later in this chapter. First however, we survey the three most widely used representations of structured data: relational, document-based and graph-based. We describe each of these over the coming sections and consider their relative suitability for representing heterogeneous data, particularly structured data.

Relational Representations

Relational representations are best known through the familiar *relational model* for database management which was first introduced by Edgar Codd at IBM Research [Cod69, Cod70] and was subsequently refined by him and others over the following two decades [Cod79, Cod90]. The relational model provides a precise specification for what a relational database should do but, deliberately, does not specify how such a relational database should be built. Most of today's database management systems, including all those which employ variants of the SQL query language, are based on ideas drawn from the relational model. Underpinning the relational model is a principled theoretical foundation that draws on the mathematical topics of set theory, first-order predicate logic, and the theory of types. For the purposes of this background chapter, the most important aspects of the relational model are relational representations of structured data. Although SQL is the *de facto* query language for accessing relational data, we do not review it here and instead refer the reader to standard texts for further details and examples [Dat91, EN06, GMN84, Mai83]. However, in "Chapter 5 – Querying and Merging Heterogeneous Data", we return to the relational model as the starting point for our own generalisation of Codd's work – giving a formal definition of both the relational model and relational algebra upon which SQL is based. In the current chapter we describe the relational representation only in sufficient depth to examine its suitability for representing important forms of structured data.

As preparation, we first introduce *propositional logic* and the closely related *attribute-value* representation that underpins the relational model. The latter also appears in discussions later in this chapter. Propositional logic allows "atomic" statements to be made that are either true or false, e.g. `person-is-male`, `age-is-21`, `age-over-30`, `room-101`. The logic also provides a set of connectives $\neg, \vee \wedge, \rightarrow, \leftrightarrow$ and punctuation symbols $(,)$ that enable atomic statements to be combined into compound statements that are either true or false. e.g.

- Person is male or female:
  `person-is-male` $\vee$ `person-is-female`
- Age is not 21:
  $\neg$ `age-is-21`
- Person is female and over 30 or person is 21:
  (`person-is-female` $\wedge$ `age-over-30`) $\vee$ `person-is-21`
- If person is not male then person is female:
  $\neg$ `person-is-male` $\rightarrow$ `person-is-female`

Attribute-value representation is similar to a subset of propositional logic in which statements consists only of atoms connected by $\wedge$ but where the values are themselves atoms rather than just true or false. For example, the propositional statement `has-name-flach` $\wedge$ `institution-bristol` $\wedge$ `person-is-male` can be represented using attribute-value representation as $\{$ (`name`, `flach`), (`institution`, `bristol`), (`gender`, `male`) $\}$. Informally, *relational data* is a data set consisting of attribute-value statements. The set of attributes, e.g. {`name`, `institution`, `gender`} in the

above example, is known as the *schema* of the data. With this preparation in place, we now move on to describe the relational representation.

A relational database is a set of relations, also known as *tables* in the literature. Expressed in Prolog syntax, a relation consists of a unique name and a pair of n-ary functors: `tuple/`n and `schema/`n. The `tuple/`n functor has the form,

```
tuple(+Value1, +Value2, ..., +ValueN)
```

where `Value1`, `Value2`, ..., `ValueN` are data values and `N` $= n$, $n \geqslant 0$. The form of the `schema/`n functor is,

```
schema(+Name1, +Name2, ..., +NameN)
```

where `Name1`, `Name2`, ..., `NameN` are all (different) unique names, also known as *attributes* in the literature. The data in the relation is defined by means of zero or more `tuple` ground facts. Each fact defines an n-tuple, also known as a *record*, in the relation. For each relation the values in the tuples are associated with a data model consisting of:

- a singleton `schema` ground fact that defines the name of the corresponding argument `Value1`, `Value2`, ..., `ValueN` in `relation/`n such that `Value1` has name `Name1`, `Value2` has name `Name2`, ..., `ValueN` has name `NameN`.
- a set of domains $\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_{\mathtt{N}}$, from which respective values `Value1`, `Value2`, ..., `ValueN` are drawn. Intuitively these domains usually consist of programming language datatypes like string, integer, float, double, date, time, etc.

In the full relational model, the above domains would be incorporated into the schema functor's arguments but they are not important in the context of this chapter and so we have omitted them for brevity in our explanation and examples.

In Example 2.2.1 we illustrate this representational scheme using a toy relational database that describes people using a pair of relations: `person` and `town`.

Example 2.2.1 *One possible relational representation of a database describing the name, gender, age, town and country of nine people.*

```
relation( person,
    schema(person_id, town_id, name, gender, age), [
    tuple(p1, t1, 'Fred'   , m, 30),
    tuple(p2, t1, 'Wilma'  , f, 30),
    tuple(p3, t1, 'Barney' , m, 26),
    tuple(p4, t1, 'Betty'  , f, 25),
    tuple(p5, t2, 'Bart'   , m, 10),
    tuple(p6, t2, 'Homer'  , m, 35),
    tuple(p7, t2, 'Marge'  , f, 32),
    tuple(p8, t3, 'Wallace', m, 50),
    tuple(p9, t3, 'Gromit' , m, 8)
]).
```

```
relation( town,
    schema(town_id, name, country), [
    tuple(t1, 'Bedrock', usa),
    tuple(t2, 'Springfield', usa),
    tuple(t3, 'Preston', uk)
]).
```

Alternative relational representations of the database in Example 2.2.1 are possible, as are other Prolog representations of course, but the chosen representation highlights a particular characteristic of well-designed relational databases. Best practice in relational representation is to design relations so that information is not duplicated. This is achieved in our example by using a separate relation for towns to avoid every person storing a copy of their town name and country as part of their tuple. Instead, each tuple in the person relation contains a reference (i.e. `town_id`) to the corresponding tuple in the town relation. When data is stored in this way it is said to be in *normalised* form (i.e. represented such that redundant duplication of values is minimised). Although this is efficient in terms of storage and can help to maintain data consistency it has the side-effect of distributing information about an individual over multiple relations. To retrieve all the data about an individual requires knowledge of the relationships between the relations (i.e. the structure of the data) so that a suitable SQL query can be defined to collect all the information together for an individual. However, the data retrieved by such a query does not itself contain information about the data's structure; that information is only held in the schema of the database. To completely reconstruct the original structure of data stored from its representation in a relational database can require consiserable effort, depending on the type and complexity of the original structure.

Representing data structures such as sets or multisets is straight forward in the relational model because it was designed to represent those structures using just a single-relation. However, when representing other types of structured data such as trees, lists and graphs it can require considerably more effort to reconstruct the original data structure from an SQL query. We illustrate the sort of complexities that arise

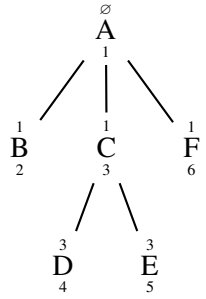
PERSON

person#	town#	name	gender	age
p1	t1	Fred	m	30
p2	t1	Wilma	f	30
p3	t1	Barney	m	26
p4	t1	Betty	f	25
p5	t2	Bart	m	10
p6	t2	Homer	m	35
p7	t2	Marge	f	32
p8	t3	Wallace	m	50
p9	t3	Gromit	m	8

TOWN

town#	name	country
t1	Bedrock	usa
t2	Springfield	usa
t3	Preston	uk

Figure 2.1: Tabular representation of the person and town relations.



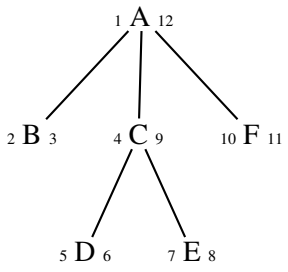
TREE

#	label	parent#
1	A	∅
2	B	1
3	C	1
4	D	3
5	E	3
6	F	1

Figure 2.2: Adjacency list representation of a tree where each node is annotated with identifiers '#' underneath and 'parent#' above the 'label'. ∅ means *no parent*.

by describing three alternative schemes for representing trees as relational data. For brevity in these examples we adopt the textbook convention of denoting relations as tables in which the title is the relation name, the column headings are the schema and the rows are the tuple values. For instance, Figure 2.1 depicts the tabular representation of our toy database from Example 2.2.1. For each of the three example schemes we show a different relational representation of the tree structure denoted by the Prolog term, `a(b,c(d,e),f)`. The first, depicted in Figure 2.2, uses an *adjacency list* structure; the second, depicted in Figure 2.3, uses *nested sets*; the third, depicted in Figure 2.4, uses a *materialised path*. These proxy structures are depicted in the examples using an annotated tree figure and a single relation, `tree`. Numerous other relational representations of trees are possible. Relational representations of other structured data, including lists and graphs, are also numerous and can similarly require considerable effort to recreate the original structure of the data when retrieving data from the database.

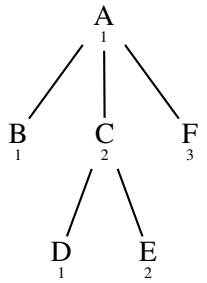
Despite the aforementioned difficulties in reconstructing the original structure of data, relational representations of structured data are widely used in practice because of the convenience offered by the relational algebra for accessing sets and multisets. Extending this convenience to data structures beyond sets and multisets is a problem that we return to in "Chapter 5 – Querying and Merging Heterogeneous Data".



TREE

#	label	left	right
1	A	1	12
2	B	2	3
3	C	4	9
4	D	5	6
5	E	7	8
6	F	10	11

Figure 2.3: Nested sets representation of a tree, where identifier '#' is an ordinal and each node is annotated with 'left' and 'right' ordinals using preorder tree traversal.



TREE

#	label	path
1	A	1
2	B	1.1
3	C	1.2
4	D	1.2.1
5	E	1.2.2
6	F	1.3

Figure 2.4: Materialised paths representation of a tree, where each node is annotated with a left-right sibling ordinal and 'path' is a string denoting its path from the root.

Document-based Representations

Semi-structured data and unstructured data are commonly referred to as *document-based representations* because they are typically stored in files which are referred to as *documents*. The ubiquitous examples from the web application domain are HTML, XML and JSON, a brief sketch of which is given in Examples 2.2.2 and 2.2.3. Although these are all textual data formats, documents are particularly awkward to represent in relational form as they are often serialisations of tree structures or, in the case of XML and JSON, arbitrary structured data with a rich range of datatypes. Indeed, relational databases tend to treat such data as atomic and opaque (if they even offer a datatype other than just `text`), which means that applications cannot easily access subparts of the data using the relational model. For example, SQL does not support queries that let a developer retrieve all the links (`<a href="...">...</a>` tags) from an HTML document in the database nor use such subparts as constraints within queries to filter or merge data; the closest that is usually possible in SQL is to use regular expressions, an approach which does not cope well with the nested nature of semi-structured data.

Example 2.2.2 (HTML/XML) *The current version of HTML, HTML5, as was its XHTML predecessor, is serialised using the XML language. A fragment of an HTML document, taken from a researcher homepage, is shown below and clearly illustrates the nested structure of the XML* elements *(tags). This structure defines a tree with root* `div`*. In fact it defines an attributed tree because, depending on the schema, nodes in the tree can have* attributes *such as the* `href` *attribute on the* `a` *element.*

```
<div>
  <h1>
    Publications by Simon Price
  </h1>
  <p>
    This is a list of all publications (co-)authored by
    <a href="/People/3133">
      Simon Price
    </a>
    that are contained in the publications database of the
    Department of Computer Science, University of Bristol.
  </p>
</div>
```

XML has an established standardised schema, XML Schema, and a range of built-in datatypes that includes most of the usual programming language datatypes [BM04]. However, there are multiple syntactic forms for representing the same data structures and, while these variants tend not to occur in HTML, they do when representing arbitrary structured data in XML. For instance, there are many ways of representing structures such as arrays and key-value objects, some of which rely on de facto *characteristics of popular XML parsers rather than adhering strictly to any standard. XML is notoriously verbose and slow to parse – particularly in the web browser environment.*

Example 2.2.3 (JSON) *JSON data consists of arrays, denoted* `[elem1,elem2,...]`*, and objects, denoted* `{"key1":elem1, "key2":elem2, ...}` *where each* `elem` *is a string, number, array, object or one of the constants* `true`*,* `false` *and* `null`*. Its simplicity has made it a popular choice for web services and document-based databases. While JSON lacks (built-in) strong typing, it is possible to represent arbitrary structured data. For instance, the example below is one possible JSON representation of our tree example from the previous section.*

```
["A", [
   ["B", null],
   ["C", [
      ["D", null],
      ["E", null]
   ]],
   ["F", null]
]]
```

Variations or extensions of the JSON syntax are not permitted under the standard. There are currently competing proposals for a standard JSON Schema but most of the candidates are themselves expressed in JSON. Different transport serialisations are used in some document-based systems, e.g. BSON (Binary JSON) is used as a compressed serialisation to reduce physical storage and bandwidth requirements. All modern web browsers have fast built-in JSON parsers.

Some large datasets involving semi-structured data (e.g. web crawls or data from high-throughput scientific instruments) now exceed the capacity of current relational database implementations. Such data is known as *Big Data* and is typically characterised by its volume, velocity and variety. In this Thesis, the heterogeneity of datasets in the domain is a central concern and so it is the *Big Variety* flavour of Big Data that is the most relevant of these, so called, Big Vs. For e-Science, e-Research and research intelligence applications where semi-structured data is commonplace, Big Variety, and Big Data in general, has led to increased interest in and adoption of document-based alternatives to traditional SQL-based relational databases. These alternatives fall into two main categories: *NoSQL databases* (or *stores*) and large-scale *distributed file systems*.

NoSQL databases Even where Big Data is not an issue, NoSQL databases are now commonly used in web applications because they simplify access to semi-structured data and subparts of that data. Two noteworthy types of document-based NoSQL systems are CouchDB and MongoDB, both of which use JSON documents as their native data records.

- **CouchDB** has a built-in web server featuring a REST API that automatically maps data to URLs so that storing and retrieving documents from a web application or as part of a workflow is trivial. Accessing subparts of JSON documents is achieved through server-side JavaScript functions which may be applied to datasets as a MapReduce task, typically extracting the required subparts and storing them as new documents in a separate dataset. Ad hoc queries are not as straight forward as in the relational model because of this need to generate what, in effect, are cached query results held as datasets.

- **MongoDB** lacks a web server but is easily integrated with one to create a REST API along similar lines to that of CouchDB. MongoDB also supports background MapReduce tasks but features an SQL-like query language which may be used to make ad hoc queries. However, queries may still result in or require the prior generation of new datasets within the database. MongoDB uses BSON (Binary JSON) as its serialisation format and is thus more efficient in terms of storage and data transfer when working with large documents.

That both CouchDB and MongoDB have built-in support for MapReduce is indicative of their Big Data focus, as is the fact that both can be distributed over multiple locations. CouchDB also supports automated replication and both systems guarantee *eventual consistency* on update operations, making them easier to scale than traditional relational database systems which offer the stronger guarantee of *ACID* (Atomicity, Consistency, Isolation, Durability).

It is worth noting here that both NoSQL and SQL databases are usually built on top of key-value stores that use b-trees as their internal data structure. The most popular open source b-tree system, BerkeleyDB (currently owned and managed by Oracle), is also our own choice for a datastore in our proof of concept software discussed in Chapter 4.

Distributed File Systems Distributed file systems have a long history in computing and are a mainstay of the enterprise computing environment. However, in the Big Data setting (and, incidentally, for many years in e-Science domains like High Energy Physics), the file system is used as a more scaleable alternative to a relational database, with pre-eminent Big Data examples being Google File System (GFS) and its open source equivalent, Hadoop File System (HFS). To the developer, these file systems appear as ordinary file systems and so are readily usable from any programming language using the same techniques as are used for local file systems. One of the appealing attributes of these systems is that developers can develop their software on a local file system, on a web server for instance, with the knowledge that it can later be scaled up if required. This was the main reason that our proof of concept software described in Chapter 3 started its life using the file system as its database.

Clearly, a file system does not normally offer the benefits of the relational model in terms of a relational algebra as a query formalism. Although there are drivers for most databases that will use the underlying file system as if it were a database, most of the desirable characteristics of the relational model are lost as a consequence of this and it is usually neither efficient nor scaleable. The Big Data approach is to use the MapReduce algorithm and leave the task of implementing the procedural steps of a query to the developer who must provide imperative code to implement the Map and the Reduce functions within the algorithm. Apache Hadoop is the most popular open source implementation of MapReduce and, by default, requires these functions to be written in Java although any language may be used. Recently, a pair of higher-level 'interfaces' to Hadoop have seen widespread adoption: Pig and Hive. Pig includes an imperative language called Pig Latin. Hive offers an SQL-like query language. Both compile their respective programs or queries into conventional Hadoop MapReduce tasks and free developers from some of the work involved.

Graph-based Representations

A *graph* is a pair (V,E) consisting of a set of vertices V and a set of edges E connecting the vertices. Examples of structures that are commonly represented as graphs include social networks, roads networks and chemical pathways as well as structures like heirarchies and trees. Structured data representing a graph is referred to as *graph-based data* and such representations are referred to as *graph-based representations*. Graph-based data can be represented in both relational and document-based forms but neither features a convenient language specifically designed for querying graph data. The Semantic Web initiative aimed to overcome this problem by introducing RDF (Resource Description Framework) [BGM04, Bec03], a standardised representation of graph data and an associated SQL-inspired query language called SPARQL. The broard objective was to create a machine-readable Web of Data along similar lines to the human-readable Web of Documents. The Web of Data is referred to as Linked Data and is highly relevant to our Thesis in the sense that data about researchers and their published works promises to simplify the task of profiling – potentially, in the future, removing the need to extract semi-structured data from web pages. Although originally envisaged in this World Wide Web context, RDF *stores* are now widely used as self-contained databases in their own right and have become yet another NoSQL middleware system available to the application developer. For a non-technical introduction to the Semantic Web, see the definitive 2001 Scientific American article [BLHL01] which launched the initiative, or for a more technical primer see [MM03].

For the purposes of this Thesis it is sufficient to understand that RDF data is, fundamentally, a representation of a directed labelled graph. To avoid the technicalities of the RDF specification we adopt the following abstract definition of an RDF graph.

Definition 2.2.4 (RDF graph) *An RDF graph is a set of statements. Each statement is an assertion about a resource and consists of a* (subject, predicate, object) *triple, where* 'subject' *is a name of the resource being described,* 'predicate' *is a name of an attribute of the resource, and* 'object' *is the attribute value. Attribute values are either resource names or typed literals.*

Representationally, an RDF graph can be a multigraph in that it may have more than one edge between a pair of vertices and consequently may actually be a multiset of statements rather than a set. However, under conventional RDF semantics [KC04], duplicate statements are ignored. As with the relational model and document-based data, logical constraints may be placed on an RDF graph using schema and ontologies which additionally embed a degree of background knowledge into the graph. The syntax of names follows that of URIs [BL98], which are a generalisation of Internet URLs such that a URI need not refer to an Internet addressable resource. In fact, a URI, despite looking like a URL, may refer to anything - including concepts, places and people. The syntax of literals is simply defined as the syntax of XML Schema datatypes [BM04] and includes all the usual types one might expect to find in a conventional programming language. One final note about our definition of an RDF graph is that we use the term *attribute* for consistency with machine learning literature but the RDF literature predominantly uses the term *property* instead.

RDF graphs have a convenient representation as a ternary relation that is particularly well suited to efficient indexing of large graphs based on the first two arguments.

```
rdf(subject, predicate, object)
e.g. rdf(person3133, firstname, 'Simon')
```

Sometimes a fourth argument is added to identify the graph. Less commonly, a binary relation is used. In which case two formulations are possible, the second of which corresponds to a multiple clause representation of the *individual centred* view of data described later in this chapter and used in our subsequent chapters.

```
predicate(subject, object)
e.g. firstname(person3133, 'Simon')

subject(predicate, object)
e.g. person3133(firstname, 'Simon')
```

One graph theoretic notation reformulates the RDF graph as a directed edge- and vertex-labelled graph $G = (V, E, l_V, l_E)$ with vertices $V = \{v_x \mid x \in S \cup O\}$ whose members are labelled by function $l_V(v_x) = x$, and with edges $E = \{e_{s,p,o} \mid (s,p,o) \in T\}$ whose members are labelled by function $l_E(e_{s,p,o}) = p$. $S = \{s \mid (s,p,o) \in T\}$, $O = \{o \mid (s,p,o) \in T\}$ and $T = \{(s,p,o) \mid \textit{statements in the RDF graph}\}$. Unfortunately, this reformulation does not quite fit the standard (V,E) graph formulation required to enable full exploitation of the wealth of existing methods from graph theory. One principled attempt to overcome this restriction is a proposal for a bipartite representation of RDF that is an intermediate model between the abstract triple syntax and the data structures used by applications [HG04].

Data that constitutes an RDF graph may be held in databases and created on demand in response to queries or it may be physically stored as a serialised RDF graph representation [Bec03], which is typically XML although other serialisation formats exist. The databases may be traditional relational databases or they may be specialised Semantic Web *triple stores* specifically designed to store and retrieve RDF data. In both cases the databases can be queried using the SPARQL query

language [PS05] in much the same way as relational databases are queried using the SQL query language. We remark that our own proof of concept software described in Chapters 3 and 4 adopts this strategy of holding data in another format but serialising it as RDF on demand. Our software from Chapter 4 also accepts RDF data as input. In this Thesis we assume document-based representations of RDF rather than relying on an RDF triple-store *per se*, noting that graph structures held in a triple store which describe an individual can be materialised using a SPARQL query to produce an RDF document.

Having surveyed relational, document-based and graph-based representations, the three most widely used representations for structured data we now turn our attention to the topic of how to compare heterogeneous data sources and structured data.

2.2.2 Comparing Heterogeneous Data

In this section we introduce three foundational topics, *database deduplication*, *record linkage* and *information retrieval*, which relate to the problem of comparing heterogeneous data, which is relevant to the profiling and matching tasks in this Thesis. Our survey of these topics ignores some traditional research community boundaries and selects a number of relevant approaches to comparing heterogeneous data that are drawn from across the collective information retrieval, machine learning and data mining literature. Before commencing our survey, we first refer the reader back to our notation from Section 2.1 (p.27) relating to the concept of a *data source* (Definition 2.1.1) that allowed us to define *profiling* (Definition 2.1.5) and *matching* (Definition 2.1.6) functions independently of specific representational formalisms in each field. Using this notation, we begin our survey by examining database deduplication, one of the most established of the topics and, as we will explain, one of the simplest examples of a match problem.

Database Deduplication

The problem of identifying approximate duplicate records in the same database is classically associated with databases that list the contact details of large numbers of people - e.g. census data, publication subscription data, patient records, customer data. Extensively researched in the statistical community since the 1950s [NKAJ59, FS69, Win99], database deduplication deals with *lexical heterogeneity*, a simplified sub-problem of the wider record linkage problem covered in the next section. Lexical heterogeneity occurs when records of the same type, describing the same entity, differ due to a textual mismatch in one or more fields. The causes of the mismatch may arise for many reasons, including:

- syntactic variations in the way the data was recorded (e.g. "P. Flach", "P Flach", "Flach, P."),
- differing degrees of detail (e.g. "P. Flach", "P. A. Flach", "Peter Flach"),
- currency of data (e.g. unmarried versus married surnames) and
- errors (e.g. typographical errors or character recognition errors).

Historically, deduplication software has been used to identify highly similar records which can then be manually checked by humans before either deleting, merging or correcting the apparent duplicates. The lexical heterogeneity problem is also encountered in data mining where deduplication is considered a part of data cleaning [Win03]. For the purposes of this Thesis we define deduplication, in terms of arbitrary data sources rather than in terms of the more usual relational databases found in the literature.

Definition 2.2.5 (Deduplication) *A data source Δ_P has elements which are descriptions $\delta(x)$ of members x of population P. A function $dedup$ is a deduplication function such that*

$$dedup(\Delta_P) = \{\delta(x) \mid \delta(x) \neq \delta(y), \delta(x) \in \Delta_P, \delta(y) \in \Delta_P\},$$

where x and y are entities in population P. When $\Delta = dedup(\Delta)$, Δ is said to be deduplicated. Fundamentally, $dedup$ reduces Δ from a multiset to a set so that the mapping from $dedup(\Delta_P)$ into P is injective.

Although by no means trivial, the database deduplication problem is the simplest case of matching and corresponds to a reflexive match against a single data source.

$$dedup(\Delta_P) \Leftrightarrow \Delta_P \approx \Delta_P.$$

Despite being a special case and a sub-problem of record linkage, the deduplication problem is most commonly addressed using the same approaches as the full record linkage problem to be described in the next section.

Record Linkage

Whereas database deduplication is concerned with identifying approximate duplicate records within the same data source, record linkage (also known as record matching or merge/purge) extends the problem to cover approximate matches across records, potentially, in multiple data sources. More significantly, record linkage deals with the case of *structural heterogeneity* where attributes in the different data sources do not necessarily have the same names and type. Furthermore, neither is there a guaranteed 1:1 mapping between attributes so that, for example, a person's address may be represented in one database as a single attribute (e.g. address) but in the other as separate attributes (e.g. street, town, city, postcode, country). This is known as the field matching problem.

The foundational work on record linkage was undertaken by the geneticist Newcombe [NKAJ59] in the 1950s, although work in this area dates back at least a decade before and much earlier still if manual, pre-computer, approaches are considered. Newcombe described how the relative frequency of an attribute value, such as a surname, among matching and non-matching records could be used to compute a score associated with matching of a pair of records. He then summed the scores of individual attribute comparisons (e.g. for the attributes first name, surname, address, etc.) to obtain an overall matching score.

The first formal mathematical theory for record linkage was introduced in the 1960s by Felligi and Sunter [FS69] of the US Bureau of Statistics. The Felligi-Sunter (FS) model compares the attributes of two records, one from each data source, and a decision is made as to whether or not the comparison pair represent the same entity, or whether there is insufficient evidence to justify either of these decisions at stipulated error levels. We adopt a definition of linkage that closely follows that of the FS model but which is expressed here in our notation.

Definition 2.2.6 (Linkage) *Two data sources Δ_P and Δ_Q have elements that are descriptions $\delta(p)$ and $\delta'(q)$ of the members of populations P and Q respectively. The set of ordered pairs*

$$\Delta_P \times \Delta_Q = \{\langle \delta(p), \delta'(q) \rangle \mid \delta(p) \in \Delta_P, \delta'(q) \in \Delta_Q\}$$

is the union of two disjoint sets

$$M = \{\langle \delta(p), \delta'(q) \rangle \mid \delta(p) = \delta'(q), \delta(p) \in \Delta_P, \delta'(q) \in \Delta_Q\}$$

and

$$U = \{\langle \delta(p), \delta'(q) \rangle \mid \delta(p) \neq \delta'(q), \delta(p) \in \Delta_P, \delta'(q) \in \Delta_Q\}$$

which are called the matched and unmatched sets respectively. A function $link$ is a linkage function such that

$$link(\Delta_P, \Delta_Q) = \langle M, U \rangle.$$

The FS model makes three decisions referred to as *linked* (P_1), *possibly linked* (P_2) and *non-linked* (P_3). Given a comparison vector γ whose components are the coded agreements and disagreements on each attribute, the probabilities of two error types (i.e. an unmatched pair assigned to P_1 and a matched pair assigned to P_3) is defined as

$$\mu = \sum_{\gamma \in \Gamma} u(\gamma) \Pr(P_1|\gamma)$$

and

$$\lambda = \sum_{\gamma \in \Gamma} m(\gamma) \Pr(P_3|\gamma)$$

where $u(\gamma)$ and $m(\gamma)$ are the probabilities of realising γ, for matched and unmatched record pairs respectively and Γ is the comparison space of possible realisations. A *linkage rule* assigns probabilities $\mathrm{P}(P_1|\gamma)$, $\mathrm{P}(P_2|\gamma)$ and $\mathrm{P}(P_3|\gamma)$ to each possible realisation of $\gamma \in \Gamma$. An optimal linkage rule $L(\mu, \lambda, \Gamma)$ is defined for each value of μ and λ that, for fixed error levels, minimises $\mathrm{P}(P_2)$. The test statistic used in the linkage rule is a monotone increasing function of the ratio $m(\gamma)/u(\gamma)$ and uses logarithms for computational convenience. Practical construction of the optimal linkage rule is simplified by an independence assumption on the attribute vector realisations $\gamma \in \Gamma$. If the independence assumption does not hold then the linkage rule will not be optimal in the above sense.

To give a concrete feel for how this is used in practice, consider the examples in Table 2.1 which show values of m and u for a subset of the attributes used in an FS-based linkage study of Florida census data [Jar89].

Table 2.1: EXAMPLE FS PARAMETERS ADAPTED FROM [Jar89]

γ *attributes*	m	u
given name	.98	.09
middle initial	.35	.03
sex - marital status	.82	.21
street name	.96	.03
house number	.99	.01

So, for the Florida census data, an agreement on *house number* carries a much higher weight than, for instance, an agreement on *middle initial*. Notice that the feature *sex - marital status* has been manually constructed prior to the linkage by combining two separate features, sex and marital status, into a single attribute. The feature selection and construction in FS-based record linkage is both manual and domain-specific.

Surprisingly, *surname* is not included in the attributes listed in Table 2.1 even though it is obviously an important factor in matching descriptions of people. The reason for this apparent omission is that, like much record linkage work, the data sources have been *blocked* into a set of smaller data sources by a first pass, matching on variables that are highly likely to cluster matching pairs. In this example, an approximate string match on surname (i.e. Knuth's SOUNDEX algorithm - so that $Smyth \sim Smith$) is used as a blocking variable prior to the calculation of μ and λ . This is an example of heuristic blocking and it is possible that a matching pair will be overlooked as it is split between blocks. Other applications have used sound blocking, where matches are guaranteed to be in the same block - although this is sometimes achieved redundantly by including a record in multiple blocks (a method sometimes referred to as *canopies*). The choice of blocking variables and heuristics is by no means a trivial problem [BCC03]. We will return to discuss this oft glossed over topic of what might be called 'preparatory phases' of record linkage shortly when we compare record linkage to our proposed semantic join.

In the 1980s, Winkler of the US Census Bureau, showed that the unknown m and u probabilities of the Felligi and Sunter model could also be calculated using the expectation maximisation (EM) algorithm [Win88, Win99] in the conditional independence case. Furthermore, Winkler went on to show that generalised EM methods could also be applied in cases where the conditional independence assumption does not hold.

A persistent problem in record linkage algorithms during the 1980s was that matches between pairs were made greedily without taking the overall best configuration of matches into consideration. Greedy matching removes the matched pair from the data sources so that subsequent matches may only select amongst the remaining individuals. Consequently, such algorithms tend to give different results if the data is presented in a different order. To overcome this problem, in 1989 Jaro introduced a linear sum assignment procedure (lsap) [Jar89] which forces a 1:1 assignment that is optimised globally over all possible pairings. Derivatives of this approach remain in

common usage for record linkage up to the present day.

Since the mid-1990s there has been a resurgence in interest in the problem of record linkage, mainly in relation to potential applications on the Web - particularly in the domains of citation matching and online social networks [GBL98, LBG99, New01, RZGSS04]. Much linkage-related research since the mid-90s simply reapplied earlier work but in a Web context. However, in the last decade, genuinely new research directions have gathered momentum. This newer work has progressed along two main lines of investigation: *similarity-based methods*, upgraded to handle structured data [GLF04, MS05, BG05, WKKH05], on the one hand and *probabilistic models*, that take account of dependencies between resolution decisions [RC04, PD04, LMR05, BG06], on the other. In many ways, this latest cohort of papers marked the start of the wider (ongoing) research effort in the machine learning community to combine statistical and logical methods.

To conclude this section we note that record linkage, in the general sense, corresponds to a match on a pair of data sources, Δ_P and Δ_Q, such that,

$$link(\Delta_P, \Delta_Q) \Leftrightarrow \Delta_P \approx \Delta_Q.$$

It is interesting to note that in existing approaches to record linkage the description function δ defines an attribute vector (or sometimes a multi-relational structure) whose components are the common attributes of the, potentially much larger, attribute vectors δ_P and δ'_Q of Δ_P and Δ_Q respectively. In other words, attributes that do not occur in both δ_P and δ'_Q are just ignored - even though absence or mismatch may in itself convey important information - particularly when considered in the context of background knowledge.

2.2.3 Information Retrieval

We briefly mention *information retrieval* (IR) here as it informs the choice of technology for our proof of concept framework implementations in later chapters of this Thesis. However, when we cast this pre-eminent research problem into our own notation it becomes clear that, for our purposes, information retrieval is equivalent to record linkage decribed in detail in the above section. The canonical task in information retrieval is, given a query in the form of a list of words (terms), rank a set of text documents in order of their similarity to the query [SJ72, FLM98]. If an information retrieval query is viewed as a singleton data source (i.e. a data source containing a single description) then the information retrieval matching problem can be viewed as a skewed version of record linkage, having exactly the same form:

$$query(\Delta_P, \Delta_Q) \Leftrightarrow link(\Delta_P, \Delta_Q) \Leftrightarrow \Delta_P \approx \Delta_Q.$$

The *vector space model*, which we describe in Section 2.3.1 later in this chapter, is a common approach to solving this problem within the field of information retrieval and is the approach we use in our own text matching work in "Chapter 3 – Workflows for Profiling and Matching Textual Content".

2.2.4 Comparing Structured Data

As machine learning and data mining has progressed from attribute-value (also known as *feature vector*) approaches towards relational representation approaches, so data matching research has progressed from comparing atomic attributes to comparing relational structures [Rae08]. This move has, at least in part, been motivated by a desire to exploit implicit semantics embedded in the intra- and inter-data structure relations: the relations within a structured object and the relations between objects. We use the term *object* here and in the rest of this section to denote a structured data record or document describing an individual.

Although this is not a machine learning or data mining thesis *per se* the frameworks discussed and developed in Chapter 4 and Chapter 5 employ the techniques described in this section and so some basic understanding of them is required. Our introduction to the topic begins not with methods for comparing structured data but with some background and terminology relating to the comparison of any data in general using the concepts of *similarity* and *distance* (often collectively referred to as *distance-based methods*, even though technically these are not identical concepts). The discussion then moves on to introduce a promising group of distance-based methods that use *kernels* to compare structured data and compute a distance from the result. Some of the material to be covered is deferred until Section 2.3 when we discuss the required background necessary to understand the Thesis; the material here is informative only but gives a more complete userstanding of our adopted method.

Similarity and Distance

To match data it is necessary to compare data to determine how well it matches. A perfect matche (equality) is the easiest to determine but for our use cases the matching is more likely to be inexact and so some method for determining an approximate match is required. An approximate match between a pair of data values indicates how alike or unalike the values are in terms of some criteria. Examples of approximate matches from the literature include: string edit distance, SOUNDEX, longest substring, synonyms and abbreviations; set cotopy, *n*-grams and various taxanomic distances such as most specific parent or least specific parent. In fact approximate matching of one sort or another forms the core of most database deduplication, record linkage and information retrieval algorithms. It therefore seems reasonable to expect approximate matching to also play an important role in matching heterogeneous data and structured data.

Beginning with some terminology, we adopt the following fairly standard definitions of similarity, (dis)similarity and distance, adapted from [EBB+04].

Definition 2.2.7 (Similarity) *Given a set $\mathcal{X}$ of entities, a similarity $\sigma : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ is a function from a pair of entities to a real number expressing the similarity between two objects such that:*

$$\forall x, y \in \mathcal{X}, \sigma(x,y) \geqslant 0 \qquad \textit{(positiveness)}$$
$$\forall x \in \mathcal{X}, \forall y, z \in \mathcal{X}, \sigma(x,x) \geqslant \sigma(y,z) \qquad \textit{(maximality)}$$
$$\forall x, y \in \mathcal{X}, \sigma(x,y) = \sigma(y,x) \qquad \textit{(symmetry)}$$

Definition 2.2.8 (Dissimilarity) *Given a set* $\mathcal{X}$ *of entities, a dissimilarity* $\delta : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ *is a function from a pair of entities to a real number such that:*

$$\begin{aligned} \forall x, y \in \mathcal{X}, \delta(x,y) &\geqslant 0 && \textit{(positiveness)} \\ \forall x \in \mathcal{X}, \forall y, z \in \mathcal{X}, \delta(x,x) &= 0 && \textit{(minimality)} \\ \forall x, y \in \mathcal{X}, \delta(x,y) &= \delta(y,x) && \textit{(symmetry)} \end{aligned}$$

Definition 2.2.9 (Distance) *Given a set* $\mathcal{X}$ *of entities, a distance (or metric)* $\delta : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ *is a dissimilarity function satisfying the triangular inequality and definiteness such that:*

$$\begin{aligned} \forall x, y, z \in \mathcal{X}, \delta(x,y) + \delta(y,z) &\geqslant \delta(x,z) && \textit{(triangular inequality)} \\ \forall x, y \in \mathcal{X}, \delta(x,y) = 0 &\Leftrightarrow x = y && \textit{(definiteness)} \end{aligned}$$

A dissimilarity function satisfying the triangular inequality but not definiteness is a pseudo-metric.

Definition 2.2.10 (Normalised (dis)similarity) *A (dis)similarity is said to be normalised if it ranges over the unit interval of real numbers* $[0..1]$. *A normalised version of similarity* σ *is denoted* $\overline{\sigma}$. *Dissimilarity* δ *respectively is denoted* $\overline{\delta}$.

Given a set $\mathcal{X}$ of entities, it trivially follows that $\forall x, y \in \mathcal{X}, \overline{\sigma}(x,y) = 1 - \overline{\delta}(x,y)$, and vice-versa

In the earliest literature approximate matches were hand-crafted and highly domain or application specific. Over time it has become progressively more common to try and learn similarity measures and metrics directly from data and, most recently, also by incorporating background knowledge expressed in logic programming languages (usually Prolog).

Distances for Structured Data

For the task of approximate matching, exploiting implicit knowledge in the structure of data is a promising technique for implementing domain independent approximate matching of complex structured objects. Two approaches stand out as being particularly relevant; both employ *kernels*.

- Convolution kernels for decomposition structures [STC04]
- Kernels for basic terms [GLF04]

Below we discuss the first of these methods. The second method is the one we adopt in our work and use in our proof of concept demonstrations in Chapter 5, hence we defer further discussion of the *kernels for basic terms* methd until we reach Section 2.2.4 in our required topics section later in the current chapter.

Kernels (see [STC04]) are a computationally convenient method for comparing the dissimilarity of pairs of objects in some feature space without having to explicitly map the objects into that space. A large number of kernels have been developed to compare all manner of object types, including logical symbols, natural numbers, sets,

lists, strings and graphs. The general form of a kernel is a function with signature $k_D : D \times D \to \mathbb{R}$ for some domain D. There is an formulaic method for computing a distance from any kernel (defined later in Definition 2.3.24) and so all kernels can be transformed into distances and thus used for approximate matching.

Example 2.2.11 (Some kernels for atomic data) *The* product kernel *(formally defined in 2.3.17) accepts a pair of inputs in $\mathbb{R}$ and multiplies them together to give the result. Obviously the inputs to the product kernel must be from a numeric domain (e.g. $\mathbb{R}$, $\mathbb{Z}$, $\mathbb{N}$) and must have a well-defined product operator. This is true of many kernels. However, assuming the existence of a well-defined equality operator on domain D then the* matching kernel *(formally defined in 2.3.16) accepts a pair of inputs from D and returns 1 if they are equal or 0 otherwise. So, trivially, product kernels $k_{\mathbb{N}}(3,4) = 12$, $k_{\mathbb{R}}(20,0.5) = 10$, and matching kernels $k_{String}("foo","bar") = 0$, $k_{String}("bar","bar") = 1$, and $k_{Colour}(Red,Red) = 1$.*

One of the interesting features of kernels is that they are closed under a range of useful operators, including addition ($+$), multiplication ($\times$), so that when valid kernels are combined using these operators then the result is also a valid kernel. Convolution kernels were introduced to aggregate the kernel comparisons of the constituent parts of a structured object into a single kernel value. The constituent parts are known as the *decomposition structure*. In Example 2.2.12 we give an informal example before giving the formal definition. Notice in the example that each pair in the decomposition structure is a value-type pair and that all the types need not be the same.

Example 2.2.12 (Decomposition structure for strings) *Let the* decomposition structure *of string* `"abc"` *be the tuple* $(("a", String), ("ab", String), ("abc", String), (3, \mathbb{N}))$ *where the first three pairs represent left substrings and the last pair represents the length of the string.*

Definition 2.2.13 (Decomposition Structure [STC04]) *A* decomposition structure *for a datatype $\mathcal{D}$ is specified by a relation $\mathcal{R}$ between an element x of $\mathcal{D}$ and a finite set of tuples of sub-components, each with an associated kernel*

$$((x_1, \kappa_1), \ldots, (x_d, \kappa_d)),$$

for various values of d. Hence

$$\mathcal{R}(((x_1, \kappa_1), \ldots, (x_d, \kappa_d)), x)$$

indicates that x can be decomposed into components $x_1, \ldots, x_d$ each with an attached kernel. Note that the kernels may vary between different decompositions of the same x, so that just as x_1 depends on the particular decomposition, so does κ_1. The relation $\mathcal{R}$ is a subset of the disjoint sum of the appropriate Cartesian product spaces. The set of all admissible partitionings *of x is defined as*

$$\mathcal{R}^{-1}(x) = \bigcup_{d=1}^{D} ((x_1, \kappa_1), \ldots, (x_d, \kappa_d)) : \mathcal{R}(((x_1, \kappa_1), \ldots, (x_d, \kappa_d)), x)$$

while the type $\mathcal{T}(\vec{x})$ *of the tuple* $\vec{x} = ((x_1, \kappa_1), \ldots, (x_d, \kappa_d))$ *is defined as*

$$\mathcal{T}(\vec{x}) = (\kappa_1, \ldots, \kappa_d).$$

The definition of a decomposition structure does not specify how one arrives at a particular decomposition and the choice of kernels for any particular decomposition in any given setting.

Before we give a formal definition, in Example 2.2.14 we show the calculation of the *convolution kernel* for a pair of strings that each have a single decomposition. Note that to keep the computation short we chose just two decompositions of our strings but in the general case, each string could have one or many different decompositions (e.g. ones in which the tuple items are permutations of the original string; or ones consisting of all the substrings of length 2) and the convolution kernel would sum the results of comparing each to arrive at the final kernel value.

Example 2.2.14 (Convolution kernel for strings) *Let the decomposition structure* $\mathcal{R}^{-1}(x) = \{\mathcal{R}(x), \mathcal{R}_2(x)\}$. *Let* $\mathcal{R}_1(\texttt{"ab"}) = (('a', Char), (2, \mathbb{N}))$, *and let* $\mathcal{R}_1(\texttt{"a"}) = (('a', Char), (1, \mathbb{N}))$. *Let* $\mathcal{R}_2(\texttt{"ab"}) = (('b', Char))$, *and* $\mathcal{R}_2(\texttt{"a"}) = (('a', Char))$. *Choose the matching kernel* κ_{Char} *for type* $Char$ *and the product kernel* $\kappa_{\mathbb{N}}$ *for type* $\mathbb{N}$. *The convolution kernel* $\kappa_{\mathcal{R}}$ *is calculated as,*

$$\begin{aligned} \kappa_{\mathcal{R}}(\texttt{"ab"}, \texttt{"a"}) &= \kappa_{Char}('a', 'a')\kappa_{\mathbb{N}}(2, 1) + \kappa_{Char}('a', 'b') \\ &= 1 \times 2 + 0 \\ &= 2. \end{aligned}$$

We now give the formal definition of *convolution kernels*.

Definition 2.2.15 (Convolution Kernels [STC04]) *Let* $\mathcal{R}$ *be a decomposition structure for a datatype* $\mathcal{D}$. *For* x *and* z *elements of* $\mathcal{D}$ *the associated convolution kernel is defined as*

$$\kappa_{\mathcal{R}} = \sum_{\vec{x} \in \mathcal{R}^{-1}(x)} \sum_{\vec{z} \in \mathcal{R}^{-1}(z)} [\mathcal{T}(\vec{x}) = \mathcal{T}(\vec{z})] \prod_{i=1}^{|\mathcal{T}(\vec{x})|} \kappa_i(x_i, z_i).$$

Also known as the $\mathcal{R}$*-convolution kernel or just the* $\mathcal{R}$*-kernel. Note that the product is only defined if the boolean expression is true, but if this is not the case the square bracket function is zero.*

While the convolution kernel has indeed been shown to be effective on structured data, we suspect that it does not sufficiently exploit the available type information for its use in matching in our setting where decompositions ideally need to be constructed automatically – making full use of strongly typed data and available background knowledge. In this respect, the higher-order logic approach of using kernels for basic terms looks more promising because it provides a principled method for constructing kernels from fundamental atomic types which may be combined to construct kernels for more complex composite types. Moreover higher-order kernels provide a declarative mechanism for incorporating background knowledge into

kernels through bespoke kernel functions as well as through kernel modifier functions that express relative importance of similarity between different parts of a data structure.

In concluding this section on comparing structured data we remark that this Thesis is not a machine learning or data mining thesis *per se* and does not depend on the particular comparison method adopted. Others [GLF04] have demonstrated that this particular method can work well on structured data in a variety of settings and this is sufficient to motivate its use in our proof of concept demonstrations as part of "Chapter 5 – Querying and Merging Heterogeneous Data".

2.3 Required Background Theory

In this section we introduce required background theory that is necessary in order to read and understand the Thesis. The prerequisite background relating to workflows, dataflows and web services has already been covered in our introduction from Chapter 1. Here we describe the *vector space model* used for profiling and matching text in "Chapter 3 – Workflows for Profiling and Matching Textual Content", the *individuals as terms* in a higher-order logic formalism that used to represent structured data throughout "Chapter 4 – Higher-Order Dataflows", and our chosen method for comparing such data in "Chapter 5 – Querying and Merging Heterogeneous Data". Finally, for those who are not already familiar with this protocol, we expand upon the overview of REST web services given in our introduction and used extensively by our proof of concept frameworks described in Chapters 4 and 5.

2.3.1 Vector Space Model

The theoretical basis for the profiling and matching functionality of our text-centric framework introduced in Chapter 3 is the well known *vector space model* from information retrieval [SWY75a]. Although this model is far from new, it has been shown to work well on general textual content across a wide range of applications – notably underpinning the widely used Apache Lucene text search engine.

The canonical task in information retrieval is, given a query in the form of a list of words (terms), rank a set of text documents D in order of their similarity to the query. The vector space model is a common approach to solving this problem. Each document $d \in D$ is represented as the multiset of terms (bag-of-words) occurring in that document. The set of distinct terms in D, vocabulary V, defines a vector space with dimensionality $|V|$ and thus each document d is represented as a vector $\vec{d}$ in this space. The query q can also be represented as a vector $\vec{q}$ in this space, assuming it shares vocabulary V. The query and a document are considered similar if the angle θ between their vectors is small. The angle can be conveniently captured by its cosine, which is equal to the dot product of the vectors scaled to unit length, giving rise to the *cosine similarity*, which is defined as follows.

Definition 2.3.1 (Cosine Similarity) *For a pair of vectors,* $\vec{q}$ *and* $\vec{d}$*, defined on the same vector space, their* cosine similarity s *is the following function.*

$$s(\vec{q},\vec{d}) = cos(\theta) = \frac{\vec{q}\cdot\vec{d}}{||\vec{q}||\cdot||\vec{d}||}$$

Although not normally a condition of the definition of cosine similarity, in the context of the vector space model the vectors are positive real valued and thus $0 \leqslant s \leqslant 1$.

However, if raw term counts are used in vectors $\vec{q}$ and $\vec{d}$ then similarity will be biased in favour of long documents and will treat all terms as equally important. The *term frequency – inverse document frequency* (tf-idf) weighting scheme compensates for this by normalising term counts within a document by the total number of terms in that document, and by penalising terms which occur in many documents.

Definition 2.3.2 (Term Frequency – Inverse Document Frequency (tf-idf)) *Let* term frequency tf_{ij} *of term* t_i *in the document* d_j *be defined as,*

$$tf_{ij} = \frac{n_{ij}}{\sum_k n_{kj}},$$

and inverse document frequency idf_i *of term* t_i *be defined as,*

$$idf_i = log_2\left(\frac{|D|}{df_i}\right),$$

where term count n_{ij} *is the number of times term* t_i *occurs in the document* d_j*, and* document frequency df_i *of term* t_i *is the number of documents in* D *in which term* t_i *occurs. Then* term frequency – inverse document frequency (tf-idf) *is defined as,*

$$\text{tf-idf}_{ij} = tf_{ij} \times idf_j.$$

In the context of our submission sifting use case, instead of comparing a single query against a set of documents, we pairwise compare every document in one collection D_1 (e.g. abstracts) with every document in another collection D_2 (e.g. reviewer bibliographies) to produce a ranked list for each document. To capture the overall importance of each term across the combined collections, df_i, and hence tf-idf_i, values are calculated over the union of both collections, $D_1 \cup D_2$.

2.3.2 Individuals as Terms

In the context of this Thesis, *individuals* are entities (i.e. "things") to be profiled or matched based on their representation in a data source. In other words, *individuals* are the elements of the underlying set described in a data source.

Definition 2.3.3 (Individual) *For a data source* Δ_P*, an* individual *is any* $x \in P$.

The *individuals as terms* representation is a generalisation of the relational model's attribute-value representation (Section 2.2.1) and collects all information about an individual into a single term. As such, the individuals as terms representation has much

in common with the various pre-relational database models that were the forerunners of the highly successful relational model which all but replaced them, until their recent resurgence in the new guise of the document-based noSQL databases and distributed file systems used in Big Data settings (Section 2.2.1). The relational model is now the *de facto* standard for database-driven applications (i.e. most business applications) but is not ideally suited for representing semi-structured data such as HTML, XML and numerous other annotated textual document formats. Neither is the relational model convenient for representing structured data such as trees, lists, graphs and so on, although the representation of such structures in relational databases is commonplace using a multitude of representations and querying patterns. As discussed in Section 2.2.1, the issue is not storing the data but in recreating its original structure upon retrieval without (sometimes) considerable effort. By constrast, the individuals as terms model is a document-based model and each individual is considered to be a self-contained document.

As a formalism, the individuals as terms model offers straight forward representations of both structured and semi-structured data while at the same time having the representational capacity to represent relations from the relational model. This representational flexibility makes the individuals as terms model a convenient representation of heterogeneous data, enabling the collection of all information about an individual in a single term irrespective of whether that information is relational, semi-structured or structured. In Example 2.3.4 we present a reworking of the toy relational database from Example 2.2.1 (p.2.2.1) to depict the same data represented using individuals as terms in Prolog notation.

Example 2.3.4 *One possible* individuals as terms *representation of the toy database describing people from Example 2.2.1, such that the individuals are chosen to be represented as the term* `person/4`*.*

```
person( name('Fred'), gender(m), age(30),
        town(name('Bedrock'), country(usa)) ).
person( name('Wilma'), gender(f), age(30),
        town(name('Bedrock'), country(usa)) ).
person( name('Barney'), gender(m), age(26),
        town(name('Bedrock'), country(usa)) ).
person( name('Betty'), gender(f), age(25),
        town(name('Bedrock'), country(usa)) ).
person( name('Homer'), gender(m), age(35),
        town(name('Springfield'), country(usa)) ).
person( name('Marge'), gender(f), age(32),
        town(name('Springfield'), country(usa)) ).
person( name('Wallace'), gender(m), age(50),
        town(name('Preston'), country(uk)) ).
person( name('Gromit'), gender(m), age(8),
        town(name('Preston'), country(uk)) ).
```

In this example we chose people (i.e. `person`) as the individuals but we could alternatively have chosen `town` or `country` or even constructed conceptual individuals such as people in specific age bands, "0-15", "16-29", "$\geqslant 30$" and so on. The choice is problem specific and different representations of the same data may be required

for different problems, each of which could require a different data structure for the individual, for example involving lists, sets, multisets or trees.

Unlike the relational representation where data is *normalised* to remove redundant duplication of data values, in the individuals as terms representation values are repeated in each term – both data values (e.g. `'Fred'`) and schema information (e.g. the `name/1` functor). Obviously this *unnormalised* form is more expensive with regard to storage but it does mean that all the information about an individual is readily available in a single term (or document if we regard terms as equivalent to documents). As well as simplifying the task of profiling and matching individuals, in physical implementations this representation has a natural one-to-one mapping with documents which, as we will discuss further in "Chapter 4 – Higher-Order Dataflows" has useful properties for parallelisation and scalability.

2.3.3 Basic Terms in a Higher-Order Logic

In the previous section we introduced the concept of *individuals as terms* using Prolog notation but in Chapters 4 and 5 we use a more expressive representation based on the higher-order logic described in this section.

The terms we use later in the Thesis for our individuals as terms representation are the *basic terms* from a family of typed terms in the higher-order logic based on Church's simple theory of types with several extensions [Chu40, Llo03]. The logic natively supports data types that are important for representing individuals, including sets, multisets, lists, trees and graphs. Strong typing helps to reduce search spaces and the type of terms provides potentially useful metadata to inform profiling and matching of individuals. The theory behind the logic and the individuals as terms formalism is set out in [Llo03]. Here we give a only brief overview and some intuition for those aspects of the logic that are relevant to the Thesis.

We begin our overview of the higher-order logic with a few technical details that are a necessary prerequisite for the definition of *basic terms*, which are the most important concept with regard to representing structured data in the Thesis. If this preamble seems a little heavy going on a first read, then perhaps skip ahead to the examples and intuitions following Definition 2.3.7 below. So, with that warning in place, we start by assuming an *alphabet* defined as follows.

Definition 2.3.5 (Alphabet [Llo03]) *An* alphabet *consists of four sets.*

1. $\mathfrak{T}$ *the set of type constructors of various arities.*
2. $\mathfrak{P}$ *the set of parameters.*
3. $\mathfrak{C}$ *the set of constants.*
4. $\mathfrak{V}$ *the set of variables.*

Included in $\mathfrak{T}$ is the constructor Ω of arity 0 with a corresponding domain of $\{True, False\}$, the booleans. Types are constructed from type constructors in $\mathfrak{T}$ and type variables in $\mathfrak{P}$ using the symbols $\rightarrow$ for function types and $\times$ for product types.

Definition 2.3.6 (Type [Llo03]) *A* type *is defined inductively as follows.*

1. *Each parameter in* $\mathfrak{P}$ *is a type.*
2. *If* T *is a type constructor in* $\mathfrak{T}$ *of arity* k *and* $\alpha_1, \ldots, \alpha_k$ *are types, then* $T\ \alpha_1 \ldots \alpha_k$ *is a type. (For* $k = 0$*, this reduces to a type constructor of arity 0 being a type.)*
3. *If* α *and* β *are types, then* $\alpha \to \beta$ *is a type.*
4. *If* $\alpha_1, \ldots, \alpha_n$ *are types, then* $\alpha_1 \times \cdots \times \alpha_n$ *is a type. (For* $n = 0$*, this reduces to* 1 *being a type.)*

A type is *closed* if it contains no parameters. $\mathfrak{S}^c$ denotes the set of all closed types obtained from an alphabet. We admit the usual nullary type constructors, including:

- Ω, the type of $\mathbb{B}$,
- *Nat*, the type of $\mathbb{N}$,
- *Int*, the type of $\mathbb{Z}$,
- *Float*, the type of floating-point numbers,
- *Real*, the type of $\mathbb{R}$,
- *Char*, the type of characters,
- *String*, the type of strings.

The set of constants $\mathfrak{C}$ includes $\top$ (true) and $\bot$ (false). A *signature* is the declared type for a constant. A constant C with signature α is often denoted $C : \alpha$. Let $[]$ be the empty list constructor with signature *List a* where a is a parameter and *List* is a type constructor. Let # be the list constructor with signature $a \to List\ a \to List\ a$.

The *terms* of the logic are the terms of typed λ-calculus and are formed in the usual way by abstraction, tupling and application from constants in $\mathfrak{C}$ and a set of variables. The set of all terms obtained from a particular alphabet is denoted $\mathfrak{L}$ and is called the *language* given by the alphabet. A basic term is the canonical representative of an equivalence class of terms [GLF04, Llo03].

Definition 2.3.7 (Basic terms [Llo03]) *The set of* basic terms*,* $\mathfrak{B}$*, is defined inductively as follows.*

1. Basic structures – *If* C *is a data constructor having signature* $\sigma_1 \to \cdots \to \sigma_n \to (T\ a_i \ldots a_k)$, $t_1, \ldots, t_n \in \mathfrak{B}\ (n \geqslant 0)$*, and* t *is* $C\ t_1 \ldots t_n \in \mathfrak{L}$*, then* $t \in \mathfrak{B}$.
2. Basic abstractions – *If* $t_1, \ldots, t_n \in \mathfrak{B}$, $s_1, \ldots, s_n \in \mathfrak{B}\ (n \geqslant 0)$, $s_0 \in \mathfrak{D}$ *and* t *is* $\lambda x.$ *if* $x = t_1$ *then* s_1 *else* $\ldots$ *if* $x = t_n$ *then* s_n *else* $s_0 \in \mathfrak{L}$, *then* $t \in \mathfrak{B}$.
3. Basic tuples – *If* $t_1, \ldots, t_n \in \mathfrak{B}\ (n \geqslant 0)$ *and* t *is* $(t_1, \ldots, t_n) \in \mathfrak{L}$*, then* $t \in \mathfrak{B}$.

The abstractions in part 2 of Definition 2.3.7 represent a key-value lookup table where t_i are the keys and s_i are the values, for $i = 1 \ldots n$, with a default value of $s_0 \in \mathfrak{D}$, where $\mathfrak{D}$ is the set of terms that do not already occur in $\{t_1, \ldots, t_n\}$. The order of the terms t_i and s_i in any basic abstraction is always the a canonical lexical total ordering, thereby ensuring that there are no semantically equivalent but syntactically different terms.

Basic abstractions merit special attention as they will be unfamiliar to many readers and yet they play an important role in later chapters of the Thesis. In essence, basic abstractions are key-value associations of type $\mathfrak{B}_{\beta \to \gamma}$, for some key of type β and value of type γ, with a default value $s_0 : \gamma$. The value t_i associated with key s_i for some basic abstractions t is $V(t\, s_i) = t_i$ for $i = 1, \ldots, n$. The set of keys s_i that occur in t is $\mathrm{supp}(t)$. With suitable choices of γ and s_0, basic abstractions can represent sets and multisets (bags).

Basic terms can represent a wide range of data structures using basic structures, basic abstractions and basic tuples, or arbitrarily nested combinations of these. In Example 2.3.8 we show lists represented as a basic terms; Example 2.3.9 shows sets; Example 2.3.10 shows multisets; and Example 2.3.11 shows a b-tree. We do not give an example for tuples as basic tuples are just n-tuples in the usual sense.

Example 2.3.8 (Lists as basic structures) *Let M be a nullary type constructor and $A, B, C, D : M$. Let $[\,]$ be the empty list constructor with signature* List *a where a is a parameter and* List *is a type constructor. Let $\#$ be the list constructor with signature $a \times \mathrm{List}\ a \to \mathrm{List}\ a$. In Figure 2.5, the lists $[A, B, C]$ and $[A, D]$ are represented as right-descending binary trees, using nested basic structures.*

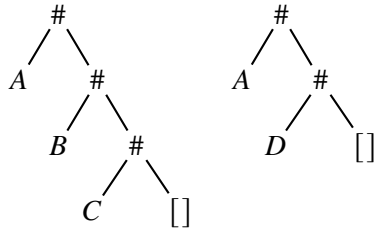


Figure 2.5: Lists $[A, B, C]$ and $[A, D]$ as basic structures.

Example 2.3.9 (Sets as basic abstractions) *A set of terms $\{A, B, C\}$ of type M can be represented extensionally as basic abstractions of type $\mathfrak{B}_{M \to \Omega}$, where Ω is the type of booleans, and the default term $s_0 = \bot$ such that,*

$$\lambda x.\ \mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else\ if}\ x = B\ \mathit{then}\ \top\ \mathit{else\ if}\ x = C\ \mathit{then}\ \top\ \mathit{else}\ \bot.$$

Example 2.3.10 (Multisets as basic abstractions) *A multiset $\langle A, A, A, B, C, C \rangle = \langle (A, 3), (B, 1), (C, 2) \rangle$ can be represented as a basic abstraction of type $\mathfrak{B}_{M \to Nat}$, where Nat is the type of natural numbers and the default term $s_0 = 0$ such that,*

$$\lambda x.\ \mathit{if}\ x = A\ \mathit{then}\ 3\ \mathit{else\ if}\ x = B\ \mathit{then}\ 1\ \mathit{else\ if}\ x = C\ \mathit{then}\ 2\ \mathit{else}\ 0.$$

Such abstractions are a mapping from each element in the multiset to its multiplicity.

Example 2.3.11 (B-trees as basic structures [Llo03]) *Let* BTree *be a nullary type constructor. Let* Null : BTree a *and* BNode : BTree $a \to a \to$ BTree $a \to$ BTree a *be data constructors where* a *is a parameter. Then* BTree a *is the type of a tree.* Null *represents the empty b-tree and* BNode *represents non-empty b-trees. For instance,* BNode (BNode Null A Null) B (BNode Null C (BNode Null D Null)) *is a basic structure representng the b-tree depicted in Figure 2.6.*

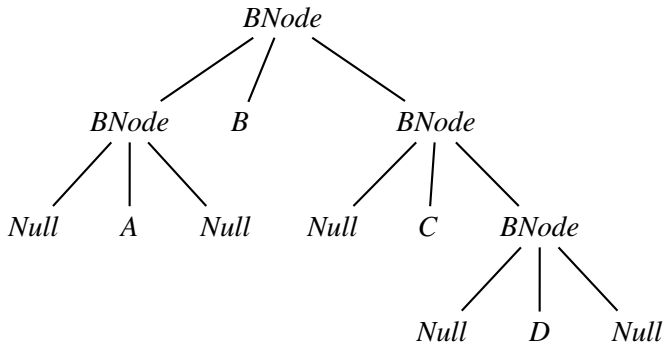


Figure 2.6: Basic term *BNode (BNode Null A Null) B (BNode Null C (BNode Null D Null))* is a basic structure representation of the depicted b-tree.

In the context of this Thesis, one important aspect of the logic is a well-defined concept of a *subterm*, which will gain significance in Chapters 4 and 5 where it becomes useful to be able to access subparts of arbitrary data structures. Informally, the way that subterms of a term are defined in the logic is to first give a definition that labels every sub-part of the term with a unique string according to a "set of rules" defined such that only legal subterms are labelled. The set of labels is called the *occurrence set* of the term and the set of rules is Definition 2.3.12. In reality the definition does not actually label the terms directly, instead induction on the structure of the term is used to generate a string consisting of a sequence of integers that records the unique path through the structure to each subterm. At the end of this recursive traversal of the structure the definition has constructed a set of these unique paths. Below we give the formal definition of an occurrence set, beginning by defining the strings that will represent the paths but the intuition can be gained just by looking at the examples which follow shortly. Let $\mathbb{Z}^+$ denote the set of positive integers and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string. $1o$ denotes the string concatenation of 1 with o and $0o$ denotes the string concatenation of 0 with o.

Definition 2.3.12 (Occurrence set [Llo03]) *The* occurrence set *of a term* t*, denoted* $\mathcal{O}(t)$*, is the set of strings in* $(\mathbb{Z}^+)^*$*, defined inductively as follows.*

1. *If* t *is a variable, then* $\mathcal{O}(t) = \{\varepsilon\}$.
2. *If* t *is a constant, then* $\mathcal{O}(t) = \{\varepsilon\}$.
3. *If* t *has the form* $\lambda x.s$*, then* $\mathcal{O}(t) = \{\varepsilon\} \cup \{1o \mid o \in \mathcal{O}(s)\}$.
4. *If* t *has the form* $(u\ v)$*, then* $\mathcal{O}(t) = \{\varepsilon\} \cup \{1o \mid o \in \mathcal{O}(u)\} \cup \{2o' \mid o' \in \mathcal{O}(v)\}$.

5. *If* t *has the form* $(t_1,\ldots,t_n)$, *then* $\mathcal{O}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{io_i \mid o_i \in \mathcal{O}(t_i)\}$.

Each $o \in \mathcal{O}(t)$ *is called an* occurrence *of* t.

Given the occurrence set of a term it is then possible to take any element of that set and use it to 'decode' the string of integers to recreate the path through the structure and identify the corresponding subterm. This, of course, relies on the same 'set of rules' for traversing the structure of the term but that is exactly what is achieved in the formal definition of a subterm, Definition 2.3.13.

Definition 2.3.13 (Subterm [Llo03]) *If* t *is a term and* $o \in \mathcal{O}(t)$ *then, the* subterm of t at occurrence o, *denoted* $t|_o$, *is defined inductively on the length of* o *as follows.*

1. *If* $o = \varepsilon$, *then* $t|_o = t$.
2. *If* $o = 1o'$, *for some* o', *and* t *has the form* $\lambda x.s$, *then* $t|_o = s|_{o'}$.
3. *If* $o = 1o'$, *for some* o', *and* t *has the form* $(u\ v)$, *then* $t|_o = u|_{o'}$.
4. *If* $o = 2o'$, *for some* o', *and* t *has the form* $(u\ v)$, *then* $t|_o = v|_{o'}$.
5. *If* $o = io'$, *for some* o', *and* t *has the form* $(t_1,\ldots,t_n)$, *then* $t|_o = t_i|_{o'}$, *for* $i = 1,\ldots,n$.

A subterm is a subterm of a term at some occurrence. A subterm is proper *if it is not at occurrence* ε.

In Example 2.3.14 we show the occurrence set and corresponding subterms of a tuple; in Example 2.3.15 we do the same for lists represented using as right-descending trees depicted in Figure 2.5 and where the left branch corresponds to a 1 and right branch corresponds to 2. More involved examples appears later in the Thesis as part of an in-depth discussion of this topic in Chapter 5.

Example 2.3.14 *If basic term* t *is the tuple* $t = (A,B,C)$ *such that* $A,B,C \in \mathfrak{B}$, *then the occurrence set of* t *is* $\mathcal{O}(t) = \{\varepsilon,1,2,3\}$, *and the subterms of* t *are indexed as* $t|_\varepsilon = (A,B,C)$, $t|_1 = A$, $t|_2 = B$, *and* $t|_3 = C$.

Example 2.3.15 *If basic terms* $s,t \in \mathfrak{B}_{List\,M}$ *are the lists* $s = [A,B,C]$ *and* $t = [A,D]$, *where* $A,B,C,D : M$, *and* M *is a nullary type constructor, then the occurrence sets of* s *and* t *are* $\mathcal{O}(s) = \{\varepsilon,1,2,21,22,221,222\}$ *and* $\mathcal{O}(t) = \{\varepsilon,1,2,21,22\}$. *The occurrence sets correspond to the subterms* $s|_\varepsilon = [A,B,C]$, $s|_1 = A$, $s|_2 = [B,C]$, $s|_{21} = B$, $s|_{22} = [C]$, $s|_{221} = C$, $s|_{222} = []$, *and* $t|_\varepsilon = [A,D]$, $t|_1 = A$, $t|_2 = [D]$, $t|_{21} = D$, $t|_{22} = []$.

As mentioned in the start of the current section, the above introduction is only an overview of the higher-order logic used in later chapters of this Thesis. One final concept which we will not expand upon in this introduction are the *higher-order* aspects of the logic, which we defer until "Chapter 4 – Higher-Order Dataflows", where the topic is explained in the context of our dataflow implementation. For now its is sufficient to know that a *higher-order function* has at least one function in either, or both, of its domain and codomain and a higher-order term is a term containing a function, and that terms in the logic can themselves include functions.

2.3.4 Kernels and Distances for Basic Terms

In this section we ignore some traditional research community boundaries and select a number of relevant approaches to comparing heterogeneous data that are drawn from across the collective information retrieval, machine learning and data mining literature.

As machine learning and data mining has progressed from attribute feature vector approaches to relational representation approaches, so data matching research has progressed from comparing atomic attributes to comparing relational structures. This move has, at least in part, been motivated by a desire to exploit implicit semantics embedded in the intra- and inter-data structure relations - the relations within a structured objects and the relations between objects. We use the term *object* here to denote a structured data record or document describing an individual.

For matching, exploiting implicit knowledge in the structure of data is a promising technique for implementing domain independent approximate matching of complex structured objects.

While the convolution kernel discussed in Section 2.2.4 has indeed been shown to be effective on structured data, we suspect that it does not sufficiently exploit the available type information for its use in matching in our setting where decompositions ideally need to be constructed automatically – making full use of strongly typed data and available background knowledge. In this respect, the higher-order logic approach of using *kernels for basic terms* looks more promising because it provides a principled method for constructing kernels from fundamental atomic types which may be combined to construct kernels for more complex composite types [GLF04]. Moreover higher-order kernels provide a declarative mechanism for incorporating background knowledge into kernels through bespoke kernel functions as well as through kernel modifier functions that express relative importance of similarity between different parts of a data structure.

By defining a *default kernel for basic terms* this higher-order logic approach enables declarative definition of customised kernels where specific tuning is required and in all other cases the default kernel for the appropriate type is applied. In common with convolution kernels, the characteristics of a particular instantiation of the default kernel must also be tuned for specific problems and datasets. However, the default kernel provides a prescriptive technique for decomposing the data based on its type structure whereas the convolution kernel provides no such guidance. The default kernel's more prescriptive recipe for kernel design, when combined with their capacity to incorporate background knowledge, has been shown to work well across a wide range of problems and data types [GLF04]. For this reason, the default kernel for basic terms is used in our proof of concept implementation in "Chapter 5 – Querying and Merging Heterogeneous Data".

As preparation for the definition of the default kernel for basic terms we first give definitions of *kernels on data constructors* and *support*. Implicit in the definition of the default kernel for basic terms is the existence of a default kernel on the data constructor of each atomic basic type. For example, a kernel on integers, a kernel on reals, a kernel on booleans, and so on. Below we give definitions for two commonly used kernels that appear in the later examples.

Definition 2.3.16 (Matching kernel) *The* matching kernel *is a function with signature* $k_\Omega : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ *such that for* $s,t \in \mathfrak{B}$,

$$k_\Omega(s,t) = \begin{cases} 1 & \text{if } s = t \\ 0 & \text{otherwise.} \end{cases}$$

The matching kernel is also known as the discrete kernel.

Definition 2.3.17 (Product kernel) *The* product kernel *is a function with signature* $k_{Real} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ *such that for* $s,t \in \mathbb{R}$,

$$k_{Real}(s,t) = st.$$

The product kernel can alternatively be defined for integers with $k_{Int} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, *or for naturals with* $k_{Nat} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$.

The same assumption of the existence of a default kernel applies to data constructors for any domain-specific atomic types, where the kernel may either be an alias for one of the aforementioned atomic basic types or a bespoke kernel function. In [GLF04] a bespoke kernel function k_{roof} is defined to compare the roof of trains such that identical roofs evaluate to 1, a flat roof compared to peak roof evaluates to 0.5 and all other combinations evaluate to 0. This captures the background knowledge that flat and peak roofs are quite similar to each other in the problem setting. Formally, kernels on data constructors are defined as follows.

Definition 2.3.18 (Kernels on Data Constructors [Llo03]) *For each type constructor* $T \in \mathfrak{T}$, *where* $\mathfrak{T}$ *is a set of type constructors,* $\kappa_T : \mathfrak{X}_T \times \mathfrak{X}_T \to \mathbb{R}$ *is a kernel on the set of data constructors* $\mathfrak{X}_T$ *associated with* T.

The definition of *support* relies on an index function V which, when applied to a basic abstraction $t \in \mathfrak{B}_{\beta\to\gamma}$ for a given 'key' $b \in \beta$, allows us to access the associated *value* $t_b \in \gamma$ in t. Intuitively, this is a way of indexing items within a basic abstraction using b as a key. For example, revisiting our earlier example of the multiset $\langle A,A,A,B,C,C\rangle$ represented as $t = \langle (A,3),(B,1),(C,2)\rangle$, a basic abstraction of type $\mathfrak{B}_{M\to Nat}$ such that,

$$t = \lambda x.\ \text{if } x = A \text{ then } 3 \text{ else if } x = B \text{ then } 1 \text{ else if } x = C \text{ then } 2 \text{ else } 0,$$

then $V(t\ A) = 3$, $V(t\ B) = 1$, $V(t\ C) = 2$ and $V(t\ b) = 0$ for all $b \notin \{A,B,C\}$.

Definition 2.3.19 (Support) *Let* $t \in \mathfrak{B}_{\beta\to\gamma}$, *for some* $\beta,\gamma \in \mathfrak{B}$. *The* support *of* t, *denoted* $supp(t)$, *is the set* $\{b \in \beta \mid V(t\ b) \notin \mathfrak{D}\}$.

In other words, the support of a basic abstraction is its set of keys (i.e. $\{A,B,C\}$ in the multiset example above).

On a technical note, our definition of support is adapted from [Llo03] which gives a more general definition in which the keys and values are not necessarily basic terms (i.e. $\beta,\gamma \in \mathfrak{S}^c$). However, in the context of this Thesis we can safely restrict our

attention to the case where $\beta, \gamma \in \mathfrak{B}$ because our data representations only require basic abstractions that map basic terms to basic terms.

With these prerequisite definitions of V and supp in place we can now define the default kernel for basic terms.

Definition 2.3.20 (Default kernel for basic terms [GLF04]) *The function* $k : \mathfrak{B} \times \mathfrak{B} \to \mathbb{R}$ *is defined inductively on the structure of terms in* $\mathfrak{B}$ *as follows.*

1. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = T\,\alpha_1 \ldots \alpha_k$, *for some* $T, \alpha_1, \ldots, \alpha_k$, *then*

$$k(s,t) = \begin{cases} \kappa_T(C,D) & \text{if } C \neq D \\ \kappa_T(C,C) + \sum_{i=1}^{n} k(s_i,t_i) & \text{otherwise} \end{cases}$$

   where s *is* $C\,s_1 \ldots s_n$ *and* t *is* $D\,t_1 \ldots t_m$.

2. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = \beta \to \gamma$, *for some* β, γ, *then*

$$k(s,t) = \sum_{\substack{u \in supp(s) \\ v \in supp(t)}} k(V(s\ u), V(t\ v)) \cdot k(u,v).$$

3. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = \alpha_1 \times \cdots \times \alpha_n$, *for some* $\alpha_1, \ldots, \alpha_n$, *then*

$$k(s,t) = \sum_{i=1}^{n} k(s_i,t_i),$$

   where s *is* $(s_1, \ldots, s_n)$ *and* t *is* $(t_1, \ldots, t_n)$.

4. *If there does not exist* $\alpha \in \mathfrak{S}^c$ *such that* $s,t \in \mathfrak{B}_\alpha$, *then* $k(s,t) = 0$.

In other words, the evaluation of the default kernel on a pair of basic terms proceeds by recursively decomposing basic structures, basic abstractions and basic tuples into subterms as prescribed in parts 1, 2 and 3 respectively, until base case atomic types are reached. As noted in the previous paragraph, there is assumed to be a kernel associated with each of these atomic types. In unwinding from the recursive descent, the value returned to each non-terminal step in the recursion is combined in the way defined in parts 1, 2 and 3 until finally a single kernel result is arrived at for the entire basic term. Examples 2.3.21 for lists, 2.3.22 for sets, and 2.3.23 for multisets (reproduced from [GLF04]) illustrate this recursive decomposition.

Example 2.3.21 (Default Kernel on Lists [GLF04]) *Let* M *be a nullary type constructor and* $A,B,C,D : M$. *Let* # *and* $[\,]$ *be the usual data constructors for lists. Choose* κ_M *and* κ_{List} *to be the matching kernel. Let* s *be the list* $[A,B,C] \in \mathfrak{B}_{List\,M}$,

$t = [A,D]$, *and* $u = [B,C]$. *Then*

$$\begin{aligned} k(s,t) &= \kappa_{List}((\#),(\#)) + k(A,A) + k([B,C],[D]) \\ &= 1 + \kappa_M(A,A) + \kappa_{List}((\#),(\#)) + k(B,D) + k([C],[]) \\ &= 1+1+1+\kappa_M(B,D) + \kappa_{List}((\#),[]) \\ &= 3+0+0 \\ &= 3. \end{aligned}$$

Similarly, $k(s,u) = 2$ *and* $k(t,u) = 3$.

Example 2.3.22 (Default Kernel on Sets [GLF04]) *Let* M *be a nullary type constructor and* $A,B,C,D : M$. *Choose* κ_M *and* κ_Ω *to be the matching kernel. Let* s *be the set* $\{A,B,C\} \in \mathfrak{B}_{M\to\Omega}$, $t = \{A,D\}$, *and* $u = \{B,C\}$. *Then*

$$\begin{aligned} k(s,t) &= k(A,A)k(\top,\top) + k(A,D)k(\top,\top) + k(B,A)k(\top,\top) \\ &\quad + k(B,D)k(\top,\top) + k(C,A)k(\top,\top) + k(C,D)k(\top,\top) \\ &= \kappa_M(A,A)\kappa_\Omega(\top,\top) + \kappa_M(A,D)\kappa_\Omega(\top,\top) + \kappa_M(B,A)\kappa_\Omega(\top,\top) \\ &\quad + \kappa_M(B,D)\kappa_\Omega(\top,\top) + \kappa_M(C,A)\kappa_\Omega(\top,\top) + \kappa_M(C,D)\kappa_\Omega(\top,\top) \\ &= \kappa_M(A,A) + \kappa_M(A,D) + \kappa_M(B,A) + \kappa_M(B,D) \\ &\quad + \kappa_M(C,A) + \kappa_M(C,D) \\ &= 1+0+0+0+0+0 \\ &= 1. \end{aligned}$$

Similarly, $k(s,u) = 2$ *and* $k(t,u) = 0$.

Example 2.3.23 (Default Kernel on Multisets [GLF04]) *Let* M *be a nullary type constructor and* $A,B,C,D : M$. *Choose* κ_M *be the matching kernel and* κ_{Nat} *to be the product kernel. Let* s *be the set* $\langle A,A,B,C,C,C\rangle \in \mathfrak{B}_{M\to Nat}$ *(i.e.* s *is the multiset containing two occurrences of* A, *one of* B, *and three of* C*),* $t = \langle A,D,D\rangle$, *and* $u = \langle B,B,C,C\rangle$. *Then*

$$\begin{aligned} k(s,t) &= k(2,1)k(A,A) + k(2,2)k(A,D) + k(1,1)k(B,A) + k(1,2)k(B,D) \\ &\quad + k(3,1)k(C,A) + k(3,2)k(C,D) \\ &= 2\times 1 + 4\times 0 + 1\times 0 + 2\times 0 + 3\times 0 + 6\times 0 \\ &= 2. \end{aligned}$$

Similarly, $k(s,u) = 8$ *and* $k(t,u) = 0$.

The name 'default' derives from the fact that in the absence of explicit bespoke kernels associated with a specific data constructor, the decomposition proceeds in the prescribed way until base case kernels are reached and each of these base cases is assumed to have a default associated kernel. It is worth noting that types associated with bespoke kernel functions need necessarily be atomic in the sense of the logic but are treated as such in the computation of the default kernel. Thus, for example, if a bespoke kernel is associated with a type that also happens to be a set (i.e. a basic

abstraction $\mathfrak{B}_{\beta\to\Omega}$), the default kernel for basic terms will not decompose the set using part 2 and will instead pass the set to the bespoke kernel to evaluate.

The relative contribution of each kernel associated with a specific type within a basic term can be modified to incorporate background knowledge into the overall kernel. This is achieved by associating a kernel modifier, of the form $\kappa_{\text{modifier}} : \mathcal{P} \times (\mathcal{X} \times \mathcal{X} \to \mathbb{R}) \times (\mathcal{X} \times \mathcal{X} \to \mathbb{R})$, with each type. Given a modifier κ_{modifier} and its parameters (an element in the parameter space $\mathcal{P}$) the input kernel is mapped to a modified output kernel. In Chapter 5 we introduce our own restated definition of the kernel for basic terms to explicitly include kernel modifiers as Definition 5.5.3, *structurally weighted default kernel for basic terms*; for the current chapter, as in the prior literature, the application of modifiers associated with specific types remains implicit. In [GLF04], various choices of modifier kernel are offered: *default*, *polynomial*, *gaussian* and *normalised*.

$$\begin{aligned}
\kappa_{\text{default}}(k)(x,x') &= k(x,x'). \\
\kappa_{\text{polynomial}}(p,l)(k)(x,x') &= (k(x,x')+l)^p. \qquad (l \geqslant 0,\ p \in \mathbb{Z}^+) \\
\kappa_{\text{gaussian}}(\gamma)(k)(x,x') &= e^{-\gamma[k(x,x)-2k(x,x')+k(x',x')]}. \qquad (\gamma > 0) \\
\kappa_{\text{normalised}}(k)(x,x') &= \frac{k(x,x')}{\sqrt{k(x,x)k(x',x')}}.
\end{aligned}$$

The *default* kernel modifier is the identity and, as its name suggests, is the default where no other modifier is specified for the kernel associated with a specific type. Given that $\sqrt{k(x,x)}$ is the norm of x in feature space, it can be seen that the *normalised* kernel is the kernelised equivalent of cosine similarity but applied to kernels rather than directly to feature vectors.

Usefully, [GLF04] proves that positive semi-definite kernels induce valid distances in the feature space and that combinations of such kernels, including the kernel for basic terms, also induce valid distances.

Definition 2.3.24 (Distances from kernels [GLF04]) *Let $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ be a kernel on $\mathcal{X}$. The distance measure induced by k is defined as*

$$d_k(x,x') = \sqrt{k(x,x) - 2k(x,x') + k(x',x')}$$

If k is a valid kernel the d_k is well behaved in that it satisfies the conditions of a pseudo-metric in Definition 2.2.9.

To complete this background chapter, below we give a brief introduction to the REST web services that were mentioned in Chapter 1 and which form a central element of our proof of concept implementations in Chapters 3 and 4.

2.3.5 RESTful Web Services

Recently, REST web services have become popular as the web API behind numerous Web 2.0 sites, including Twitter, Flickr and Facebook and are also widely used in e-Science and e-Research. Like conventional websites, RESTful services offer a state-

less, cacheable, layered and uniform client-server interface. However, unlike conventional sites, which are designed to render human-readable data as HTML pages to be viewed in a browser, RESTful sites serve data in formats such as XML, JSON and RDF, that may be readily consumed by arbitrary applications. Furthermore, in the same way that HTML forms on conventional web pages can be used to submit data from the client to the server for storage and processing, arbitrary RESTful applications can use exactly the same protocols to achieve the same end. Also, the usual HTTP authentication and authorisation mechanisms can be used to control access to specific services and resources.

The intuition behind REST is that URIs are used to represent resources and that HTTP request methods are used to specify fundamental operations on those resources. The most widely used HTTP request method, `GET`, is invoked every time a web browser requests a URI from a web server. HTTP `GET` is only one of several HTTP request methods; the five that are most significant for REST web services are summarised in Table 2.2. These correspond to the *CRUD(E)* (Create, Replace, Update, Delete and Exists) operations from Web 2.0. Additional operations are specified by adding verbs into the URIs themselves. Pairs of attribute-value parameters can be supplied with the HTTP request in the usual way to modify the behaviour of operations. The content type of the result is normally specified by either content negotiation embedded in the HTTP header or by simply adding a filename extension to the url (e.g. `.json` for JSON or `.xml` for XML).

Table 2.2: INTERPRETATION OF HTTP REQUEST METHODS IN REST

Method	Usage in REST	Read-only
`GET`	*show* and *list*	yes
`HEAD`	*exists* (resource exists?)	yes
`POST`	*create* and *compute*	no
`PUT`	*update* and *recompute*	no
`DELETE`	*destroy*	no

The following is an example of a REST request, using `curl`[1] as a convenient command line client, to retrieve a (usually JSON or XML by default) representation of a book by its ISBN.

```
curl -X GET http://www.foo.com/books/0471941522
```

A similar request to the same URI using the `DELETE` method will delete that same representation.

```
curl -X DELETE http://www.foo.com/books/0471941522
```

[1] cURL – `http://curl.haxx.se`, visited April 2014.

As the potentially destructive nature of the second example should make obvious, authentication and authorisation are sometimes required for REST operations. All the usual HTTP techniques can be used for this purpose. For instance, REST operations return standard HTTP status codes (e.g. `200 OK`, `404 Not Found` and so on). Armed with this knowledge, a Web 2.0 application developer has all the information they need to start developing with REST.

The RESTful services and the workflows presented in this Thesis are compatible with workflow management systems such as Kepler and Taverna [ABJ+04, HWS+06]. Compatibility would be further increased by the creation of a WSDL 2.0 description of their REST APIs which would add SOAP support to simplify integration into existing workflow design and enactment tools [LGS07, SGL07].

2.4 Summary

Some of the necessary background for this Thesis has already been covered in our introduction from Chapter 1. In this chapter we introduced *terminology* used throught the Thesis and which is relevant to all readers. The remainder of the chapter was divided in two: the first part surveyed *informative background topics* relevant to our work but about which readers only require a general awareness; the second introduced the *required background theory* necessary to read and understand this Thesis.

Chapter 3

Workflows for Profiling and Matching Textual Content

In this chapter we explore different solutions to the *submission sifting* problem of matching submitted conference or journal papers to potential peer reviewers based on the similarity between the paper's abstract and the reviewer's publications as found in online bibliographic databases. We describe how the use of e-Science inspired workflows and components resulted in the SubSift framework for profiling and matching general textual content from documents and web pages. We introduce SubSift through progressive generalisation of a real-world case study of a web application to support academic peer review for machine learning and data mining conferences [FSG + 09, PFS10a, PFS10b].

After describing different implementations of SubSift, we investigate its application to other research intelligence use cases by incorporating the same SubSift profiling and matching model into re-usable web services and workflows. This investigation demonstrates the utility of the workflow approach to profiling and matching general textual content. Experience from the re-use of SubSift in different settings suggests the possibility of a more flexible approach to addressing these and, potentially, a much wider range of web-centric use cases [PFS + 10d, PFS + 13, PF13b].

3.1 SubSift Case Study

SubSift originated as a set of bespoke tools to support the research paper review process of a major data mining conference, *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining 2009* (SIGKDD'09).

Experiences with SubSift Tools

The initial SubSift tools were conceived by the SIGKDD'09 Programme Co-Chairs and written by members of the local organising group with external collaborators at Microsoft Research, using a mixture of Prolog, Java, C++, C# and Matlab. The text analysis tools that contribute to the subject of this chapter built upon the prior work of Spiegler, one of the local group members [Spi06]. The emphasis in the development of these tools was on hitting the tight deadlines of the conference paper peer review

process rather than on producing generic re-usable software. As reported in Flach et al. [FSG + 09] and summarised below, the deadlines were met and the SubSift tools delivered the required functionality, assisting in the allocation of 537 submitted research papers to 199 reviewers.

Using these tools, each reviewer's bids were initialised based on subject areas (keywords) as well as a textual comparison between the paper's abstract and the reviewer's publication titles as listed in the DBLP computer science online bibliography [Ley02]. The textual comparison treated the abstract text and the reviewer's publication titles as bags of words. After first removing commonly occurring words ('stop-words'), the number of occurrences of each word were counted and normalised based on term frequency-inverse document frequency (tf-idf) (Definition 2.3.2) [SJ72]. The resultant tf-idf weighted term vectors were then compared by calculating their cosine similarity (Definition 2.3.1), a well-known technique borrowed from the vector space model in information retrieval [SWY75b]. In a final step, these similarity scores were combined with counts of the number of subject areas in common using a weighted sum, with manually chosen weights, to arrive at a combined similarity score.

The combined similarity scores were discretised into four bins using manually chosen thresholds, with the first bin being a 0 (no-bid) and the other three being bids of increasing strength: 1 (at a pinch), 2 (willing) and 3 (eager). These initial bids were exported from SubSift and imported into the conference management tool[1]. On average, each reviewer's bids were initialised with approximately 7.5 papers (1.4% of 537) at bid level 3, 16.6 (3.1%) at level 2, and 52 (9.7%) at level 1, leaving the remaining 460.9 (85.8%) papers at level 0, no-bid.

Based on the same similarity information, each reviewer was sent an email containing a link to a personalised SubSift generated web page listing details of all 537 papers ordered by a choice of: initial bid allocation, keyword matches, and similarity to their own published works. The page also listed the keywords extracted from the reviewer's own publications and those from each of the submitted papers. Guided by this personalised perspective, plus the usual titles and abstracts, reviewers affirmed or revised their bids recorded in the conference management tool.

Qualitatively, comments received from reviewers at this stage ranged from a surprised, "*There are already non-default bids on my papers in the system (and they are not bad)*" via a contented, "*I reviewed my automatically assigned papers a few days ago and was very happy with them so I did not put any bids in*" to an excited, "*as I go thru my paper assignments, I am extremely impressed by quality of your initial automated assignment!*".

To quantitatively evaluate the performance of the SubSift tools, the bids made by reviewers were considered to be the 'correct assignments' against which SubSift's automated assignments were compared. Disregarding the level of bid, a median[2] of 88.2% of the papers recommended by SubSift were subsequently included in the reviewers' own bids (precision). Furthermore, a median of 80.0% of the papers on which reviewers bid for were ones initially recommended to them by SubSift (recall). Combined, as the harmonic mean of precision and recall, this gives an F-measure of

[1] Microsoft CMT – `http://cmt.research.microsoft.com/`, visited April 2014.

[2] Median precision and recall are quoted because of the skew of the distribution. However, some other authors use the mean. For the sake of comparison, these were 58% and 69% respectively.

72.7%. These results suggest that the papers eventually bid on by reviewers were largely drawn from those that were assigned non-zero bids by SubSift.

Although not mentioned in the paper, these results are comparable with similar published results using language models and are all the more impressive for being on real-world data in a practical setting [DKL08, MM07, HP06]. Overall, both qualitative and quantitative results were very encouraging and demonstrated that there was a lot of scope for automated support during the bidding process driven by text analysis.

The case for SubSift Services

The promising results from SIGKDD'09 also suggested that the SubSift tools might be used to support submission sifting for other conferences or workshops and, potentially, be applied to the remaining use cases described in this Thesis. However, the short timeframe in which the tools were developed and their focus on a single conference inevitably resulted in a number of obstacles to their reuse. To overcome these, the author of this Thesis submitted a project proposal to the UK Joint Information Services Council (JISC) under the Rapid Innovation strand of the Information Environment Programme 2009-11, which was subsequently awarded. The project, called *SubSift Services*, took an e-Science web services approach to addressing the obstacles to the re-use of SubSift tools [PFS10a, PFS10b]. Below, we list the main obstacles and, in each case, briefly explain how the SubSift Services project addressed them.

- Usage was via the command line only; there was no web or graphical user interface. SubSift Services repackaged the software as both a website and as a family of web services.

- The tools enforced a specific workflow; it was not possible to "pick and mix" the most useful features on an application-by-application basis. SubSift Services' web services allowed compositions or "mash-ups" of features to create workflows to support bespoke applications.

- The tools software was largely undocumented and not packaged up for redistribution. Re-use of any of the tools would probably only be possible with the help of the original developer. The project documented, packaged and released the bid initialisation and paper assignment software as an open source application via the Google Code website.

- Customisation required modification of the software. SubSift Services enabled customisation of the software through well documented configuration settings and through support for mash-ups with tools like Yahoo! Pipes.

- The list of potential reviewers required a bespoke file format. SubSift Services adopted standards-based file and data formats.

- The DBLP computer science online bibliography was the only bibliographic data source supported. SubSift Services accepted input from different bibliographic data sources, such as Citeseer, Google Scholar, eprints, homepages and blogs.

- Text acquisition for submitted papers was hard-coded to come from a single data source. SubSift Services accepted abstracts from a wider range of sources; they did not, however, directly integrate with conference management systems as part of the project.

- Dissemination beyond the immediate research community was unlikely. The SubSift Services project also promoted and disseminated the SubSift tools' functionality to the wider academic community.

As well as re-implementing the decomposed functionality of SubSift Tools as a family of web service components, the project also sought to evaluate their application to other conferences and to explore their re-use in different contexts elsewhere as a generalised submission sifting workflow.

Choice of Technology

At its outset the SubSift Services project faced a number of technology choices, the most important of which we list below along with the rationale for our choices.

1. *Whether to use an existing information retrieval system.*
   Information retrieval (IR) systems are designed to accept a query in the form of a list of words (terms) and rank a set of text documents in order of their similarity to the query (see Section 2.2.3). Existing IR systems such as Apache Lucene[3], recent versions of PostgreSQL[4] and Oracle Ultra Search[5] all support text matching on large-scale document collections. For submission sifting, instead of comparing a single query against a set of documents, we pairwise compare every document in one data source Δ_P (e.g. abstracts) with every document in another data source Δ_Q (e.g. reviewer bibliographies) to produce a ranked list for each document (or conversely for each reviewer). Clearly, submission sifting could be implemented by iterating over all the documents in each data source and using an IR system to perform the necessary text matching $query(\Delta_P, \Delta_Q)$ to retrieve a ranked lists of matches for each document, echoing the form of record linkage functions $link(\Delta_P, \Delta_Q)$ from Section 2.2.2,

   $$query(\Delta_P, \Delta_Q) \Leftrightarrow link(\Delta_P, \Delta_Q) \Leftrightarrow \Delta_P \approx \Delta_Q.$$

   However, despite being the key feature of SubSift Services, we suspected that the amount of code involved in text matching represented only a modest part of the overall functionality of the proposed web services. We anticipated that most of the system's code would be concerned with implementing web services to *manage* (e.g. create, read, update, delete, manage access) of the documents to be pmatched or in code to *control* processing (e.g. initiatiating imports, web harvesting, profiling, matching, report generation). Crucially, we also wanted to be able to record and expose metadata describing the factors that

[3] Lucene – `http://lucene.apache.org`, visited April 2014.

[4] PostgreSQL – `http://www.postgresql.org`, visited April 2014.

[5] Ultra Search – `http://docs.oracle.com/cd/B13789_01/ultra.101/b10731/over.htm`, visited April 2014.

contributed to a particular match (e.g. the relative contribution of different terms to an overall similarity score) and, intuitively, it seemed easier to both record and access that metadata by having low-level access to the matching code – without having to patch or integrate with an existing open source system like Lucene in ways that, although possible, would be unsustainable beyond the life of the project because new versions of the tool would require updates to the patch. So, on balance we opted for a compromise and used an existing low-level information retrieval library[6] rather than a more sophisticated information retrieval system.

2. *Whether to connect service components using an existing workflow system.*
   The SubSift Services project aimed to produce a family of web service components that could be used in a variety of workflows such as those in our motivating use cases. As discussed in Section 1.2.2, there are a number of established workflow systems, including e-Science engines like Taverna, RapidMiner and Kepler as well as numerous general purpose integration tools based around BPEL. Most of these systems allow web service components to be invoked as part of their workflows and also offer a choice of different local and Internet communication protocols to suit different application settings. Such systems were clearly strong technology candidates for defining and enacting workflows built from the planned SubSift Services. However, we wanted to the explore the idea of enacting submission sifting and our other workflows through client-side web applications, invoking SubSift Services directly from the browser, rather than via an (already well-proven) server-side workflow engine; we were in no doubt that our workflows could be enacted using established workflow systems. Our motivation for our approach was that, if successful, it would enable third-party users of SubSift Services to "mash-up" new workflows themselves without requiring server-side code changes on our part and without requiring their own installation of a workflow engine. Had the choice been made purely on the grounds of software engineering best practice and cost effectiveness then using an established workflow system would have made more sense.

3. *Which protocol to base the web services on.*
   Unpicking this question, there are two main considerations: data transmission and service invocation. One important idea behind developing SubSift Services as a family of services (i.e. a collection of closely interoperating services), rather than as entirely independent services that could each be used without the others, was to minimise the need for data transmission by allowing the services to share a common local (to the services) data store. This meant that a service to *match* textual documents could defer the creation, update and deletion of those documents in the data store to a separate *documents* service and that retransmission of documents from the client to the server is not required at each processing step – only metadata identifying the relevant documents in the store is required in order for the match service to process that data. Consequently, so long as data can be ingested in a reasonable time then the choice of data transmission protocol was not the most important consideration for the project – thereby making the choice of service invocation our main consideration. As

[6] Perl CPAN modules – `Text::Document` and `Text::DocumentCollection`.

discussed in Section 1.2.2, the dominant protocol for web services is SOAP-based message passing but for SubSift Services we opted for the less popular option of REST web services in combination with their closely entwined with the HTTP protocol. Had we opted to implement SubSift Services using an existing workflow system then SOAP-based message passing would have been a more natural choice. Our choice was based on REST's relative ease of use from client-side code in the browser and the fact that, in our experience, more web developers are familiar with REST than SOAP (possibly because of the latter's popularity in Web 2.0 applications).

Later, in the chapter summary we revisit these decisions in light of our subsequent experiences of developing using SubSift web services to implement the full range of workflows introduced in our motivating use cases.

Generalised Submission Sifting

Recall from our description of "Use Case 1 – Submission Sifting" (p.2) that submission sifting can be divided into three specific problems:

1. matching submissions to reviewers;
2. ranking potential assignments;
3. allocating papers to reviewers.

The approach taken in SubSift Tools addresses all three of these problems by reducing them to the single task of computing pairwise similarity between a set of representations of the reviewers and a set of representations of the papers. For problem 1, "*matching submissions to reviewers*", ranking pairwise similarity scores returned by SubSift enables the discretisation of similarity scores into the required number of initial bid bins to assign initial bids for every reviewer-paper pair. These pairs can be grouped by reviewer to give a list of papers ranked by similarity to each reviewer. This list solves problem 2, "*ranking potential assignments*", by assisting reviewers in their revision of SubSift-initialised bids. Conversely, solving problem 3, "*allocating papers to reviewers*", requires that the pairs be grouped by paper to give a list of reviewers ranked by similarity to each paper. This list assists the programme co-chair in allocating papers manually where required.

SubSift's solution to this task can be generalised to define a *generalised submission sifting workflow* composed from a set of flexible web service components. In the first step towards this goal we will define the inputs and outputs of SubSift, before going on to decompose the high-level task into a number of smaller steps that are well suited to implementation as web services. The final step before arriving at our goal is to parameterise these components to introduce the flexibility required to support our other use cases.

Throughout the remainder of this case study we use Prolog syntax because it allows concise expression of the behaviour of SubSift and because its proximity to first-order logic leads naturally into our later development of the concepts introduced here. So, in the first instance, we can describe SubSift as a predicate, `subsift/3`, with two inputs and one output, with the following signature.

Definition 3.1.1 (Submission Sifting Signature) *Submission sifting,* `subsift/3`*, is a predicate with signature,* `subsift(+Reviewers, +Papers, -Matches)`.

The set of n reviewers and the set of m papers can be represented as $n+m$ Prolog ground facts using the predicates `reviewer/2` and `paper/2` respectively. Each `reviewer(Rid, Rvalue)` fact associates some unique identifier `Rid` with some ground first-order representation, `Rvalue`, of the reviewer. Each `paper(Pid, Pvalue)` fact associates some unique identifier `Pid` with some ground first-order representation, `Pvalue`, of the paper. The representation used by the initial SubSift tools was to represent: a reviewer by the string formed by concatenating all their publication titles together; and a paper by the string formed by concatenating its title and abstract. Reviewer names were used as identifiers; papers used their existing unique identifiers from the conference management tool. Abbreviated data from a recent SIGKDD conference with 300 reviewers and 590 papers is shown below. Notice that, to preserve confidentiality, only accepted papers are shown in our examples and, for the same reason, similarity scores have been changed.

```
reviewer('A Tan', "Creating an Immersive Game World with...").
reviewer('A Hinneburg', "Ranked Set Search in Medline...").
reviewer('A Evfimievski', "Epistemic privacy. Epistemic...").
⋮
reviewer('Z Zhang', "A Multiple-Instance Learning Based...").

paper(p1, "Clustering by Synchronization|Synchronization is...").
paper(p2, "Inferring Networks of Diffusion and Influence...").
paper(p3, "Unifying Dependent Clustering and Disparate...").
⋮
paper(p590, "Versatile Publishing for Privacy Preservation...").
```

Assuming that a list of all `reviewer/2` facts are collected into the variable `Reviewers` and a list of all `paper/2` facts are collected into the variable `Papers` then,

```
?- subsift(Reviewers, Papers, Matches).
```

will unify `Matches` with a list of $n \times m$ facts, `match(Rid, Pid, Similarity)`, where each `match/3` fact represents the cosine similarity score of a pair of tf-idf term-weight vector representations of values, `Rvalue` and `Pvalue`, from the Cartesian product of `Reviewers` with `Papers`. For our SIGKDD example, `Matches` would be unified with a list of facts as depicted below.

```
match('A Tan', p1, 0.005).
match('A Tan', p2, 0.012).
match('A Tan', p3, 0.012).
⋮
match('A Tan', p590, 0.000).
match('A Hinneburg', p1, 0.006).
match('A Hinneburg', p2, 0.001).
match('A Hinneburg', p3, 0.000).
```

```
⋮
⋮
match('Z Zhang', p1, 0.000).
match('Z Zhang', p2, 0.001).
match('Z Zhang', p3, 0.218).
⋮
match('Z Zhang', p590, 0.011).
```

This representation of what is essentially an $n \times m$ similarity matrix may seem rather wasteful, having a `match/3` fact for each element in the matrix (or at least for half of the matrix if we exploit the symmetry of the similarity relation). However, representing matches in this way allows an arbitrary amount of additional information about each pairwise match to be recorded within the same predicate. We will elaborate on the nature of this extra information later in this section.

Having established the inputs and outputs of `subsift/3` we now turn to the question of how to decompose the functionality of this high-level predicate into a series of lower-level predicates that are amenable to implementation as a collection of reusable web services. To explore the web service requirements for submission sifting we re-implemented the functionality of SubSift Tools as a single integrated Web 2.0 application. This prototype SubSift web application was iteratively refined during the peer review of the 2010 SIAM International Conference on Data Mining (SIAM DM'10), guided by feedback from the programme chair, Bart Goethals. The goal was to identify functional components within the (hard-coded) workflow for subsequent isolation and repackaging as web services. This rapid application development and its iterative refinement led to the following workflow.

- The programme chair pasted the names of the 145 programme committee members into the SubSift application (Figure 3.1) and worked through a disambiguation process in which SubSift searched the DBLP online bibliography for each of the names, presenting ambiguities to Goethals for resolution, to establish the correct DBLP author page for each Programme Committee (PC) member.

- The programme chair uploaded a file containing the 327 submitted abstracts – as exported directly from the conference management tool (i.e. the same Microsoft CMT service used at SIGKDD'09).

- SubSift then compared the PC members' online bibliographies, harvested from the DBLP website, with each of the submitted abstracts and produced personalised web pages, listing the submitted papers most closely matching each PC member (submission sifting problem 2).

- The programme chair typed in threshold values for the resulting similarity scores to discretise them into initial default bids, Figure 3.2, repeating the process until he was satisfied with the assignments (submission sifting problem 1). To accelerate the process of manually choosing thresholds, Goethals exported the entire similarity matrix as a CSV file to analyse in a spreadsheet.

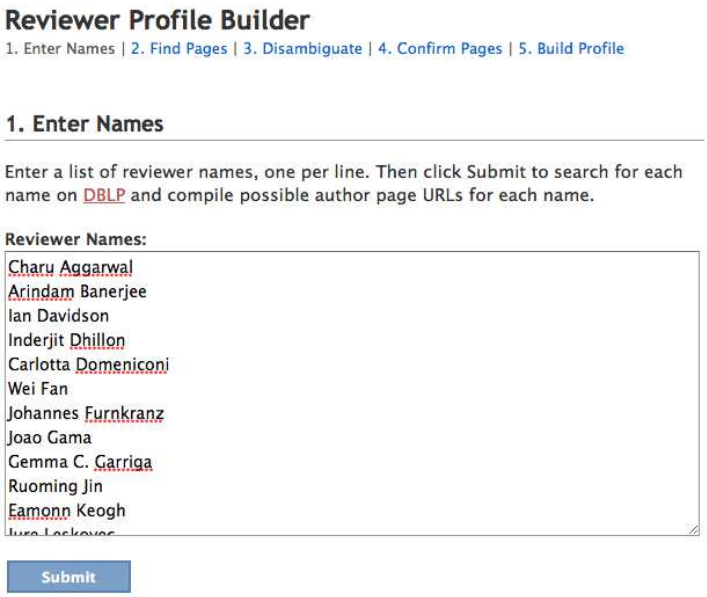


Figure 3.1: Enter Names step of Reviewer Profile Builder in SubSift prototype.

- The personalised web pages were made available to the PC members to guide their bidding for papers to review. Also, to assist Goethals in assigning papers that no one bid on, SubSift produced a list of the closest matching reviewers for each paper (submission sifting problem 3).

The overall workflow structure that we eventually arrived at can be seen in the top-level menu of the prototype SubSift web application, Figure 3.3.

Abstracted somewhat, and ignoring some of the details for now, the prototype's overall submission sifting workflow structure can be translated to the following definition of the `subsift/3` predicate. This is the primary definition that most clearly expresses the profile-match paradigm upon which SubSift is founded. Later we will progressively introduce lower-level details into the predicate's definition, with the unavoidable side-effect of gradually obscuring this higher-level structure, and so it is worth stressing that this is the most concise and intuitive definition of SubSift.

Definition 3.1.2 (Submission Sifting) submission sifting *is a predicate* `subsift/3` *with the signature and definition,*

```
subsift(+Reviewers, +Papers, -Matches) :-
    profile(Reviewers, P1),
    profile(Papers, P2),
    match(P1, P2, Matches).
```

where input variables `Reviewers` *and* `Papers` *are unified with the set of representations of reviewers and set of representations of papers respectively. Output variable* `Match` *is unified with a list of reviewer-paper pairs ranked by descending similarity.*

Note that, unlike in Prolog, here there is no procedural significance to the ordering of the clauses in the body of `subsift`. So, for instance, there is no reason why the

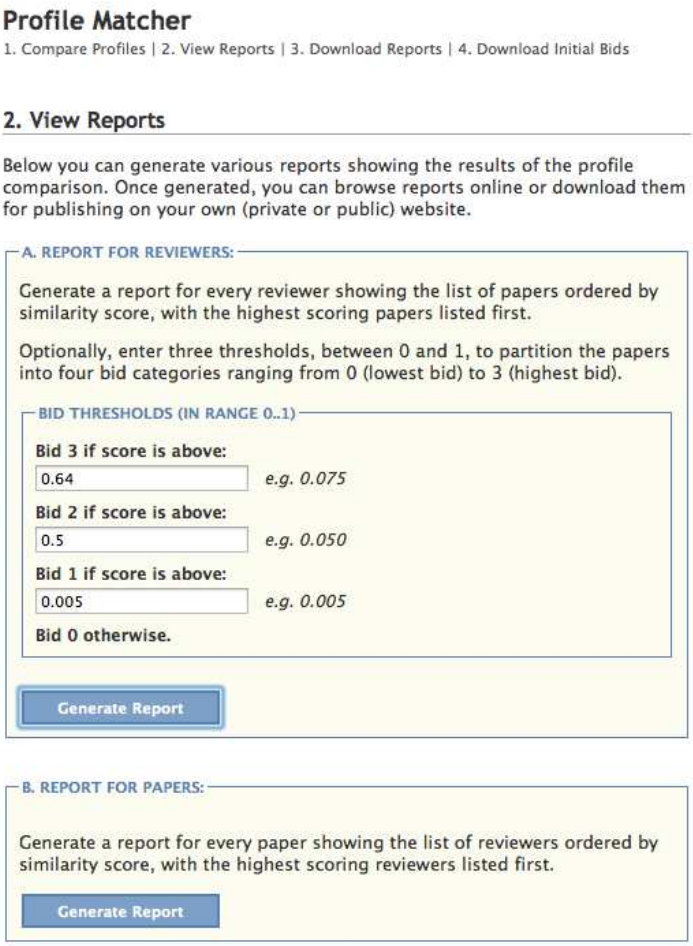


Figure 3.2: View Reports step of Profile Matcher in SubSift prototype.

two `profile` predicates could not be transposed. This principle holds for all the subsequent refinements of `subsift` in this chapter.

Both reviewers and papers are represented by textual documents of some kind: reviewers by the concatenated titles of their publications on DBLP; papers by the concatenation of their title and abstract. As a practical consideration it is useful to map both the list of reviewers and the list of papers into a common format prior to profiling and matching. This incorporates into `subsift/3` some of the preamble code that would otherwise be required prior to its invocation and also simplifies the implementation of `profile/2` by removing the need to handle multiple input types. So, as an intermediate step towards generalised submission sifting, we update the body of `subsift/3` to include two `document/2` predicates as follows.

```
subsift(+Reviewers, +Papers, -Matches) :-
    document(Reviewers, D1),
    document(Papers, D2),
    profile(D1, P1),
    profile(D2, P2),
    match(P1, P2, Matches).
```

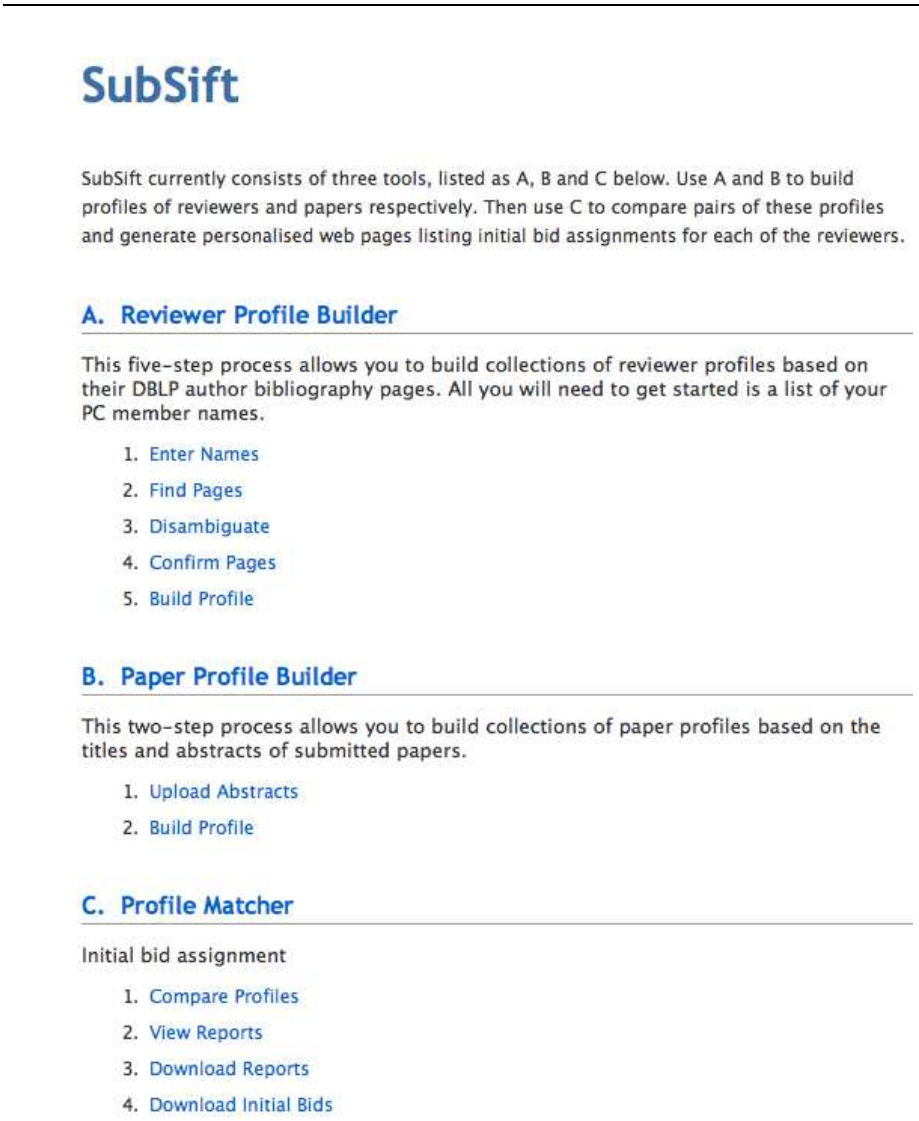


Figure 3.3: Top-level menu depicting workflow of SubSift prototype.

The role of `document/2` is to transform the input list of `reviewer/2` or `paper/2` terms into a standardised `documents/2` structure, where the first argument is a unique identifier and the second argument is a list of `item/2` terms as shown below.

```
documents(
    reviewers, [
        item('A Tan', "Creating an Immersive Game World with..."),
        item('A Hinneburg', "Ranked Set Search in Medline..."),
        item('A Evfimievski', "Epistemic privacy. Epistemic..."),
        ⋮
    ]).

documents(
    papers, [
        item(p1, "Clustering by Synchronization|Synchronizat..."),
        item(p2, "Inferring Networks of Diffusion and Influe..."),
```

```
        item(p3, "Unifying Dependent Clustering and Disparate..."),
        ⋮
    ]).
```

Treating both reviewers and papers as instances of `document/2` enables their profiling to both be performed by `profile/2`, a predicate that transforms each string in the second argument of each `item(Id, String)` into its vector space representation as a tf-idf term-weight vector. To sparsely represent this vector SubSift uses a bag-of-words (multiset) representation that only records non-empty elements of the vector and which we emulate in Prolog syntax as illustrated by the following example `profiles/3` predicate.

```
profiles(
    reviewers, [
        item('A Tan', [
            term( name('self organizing'),
                n(18),
                tf(0.00201274740020127),
                idf(4.8073549220576),
                tfidf(0.00967599112121624)),
            term( name('organizing neural'),
                n(12),
                tf(0.00134183160013418),
                idf(6.8073549220576),
                tfidf(0.00913432394774586)),
            term( name(organizing),
                n(19),
                tf(0.00212456670021246),
                idf(3.63742992061529),
                tfidf(0.00772796248369569)),
            ⋮
        ]),
        item('A Hinneburg', terms(...)),
        item('A Evfimievski', terms(...)),
        ⋮
    ],
    [
      term(dt(169), n(2581), name(mining)),
      term(dt(168), n(1186), name(analysis)),
      term(dt(160), n(863), name(approach)),
      term(dt(160), n(1083), name(systems)),
      term(dt(159), n(1980), name(learning)),
      term(dt(155), n(1001), name(clustering)),
      ⋮
    ]).
```

As can be seen from the profile `item` for reviewer `'A Tan'`, each term is accompanied by `n`, the number of occurrences in the document, the `tf`, `idf` and `tfidf`, the term frequency, inverse document frequency and their product respectively. The calculation of these statistics relies on first computing, for each term in the combined vocabulary of all reviewer documents, the `dt` count of number of documents that the

term occurred in and `n`, the total number of occurrences. These corpus statistics form are represented as a list of terms in the third argument of `profiles/3` above.

The final stage of the `subsift/3` definition is `match(+P1, +P2, -Matches)`, which given a pair of profiles `P1` and `P2`, unifies `Matches` with a list of `match/3` pairwise similarities, e.g. `match('A Tan', p3, 0.012)`. However, to generate useful reports as part of a submission sifting application and to support our intended wider range of use cases, it is helpful to provide additional information about each match:

- to return a discretised version of the similarity as a bid value, e.g. `bid(3)`
- to record the relative contribution of each term towards the cosine similarity so that this can be displayed within a web application, e.g.

```
[
  term( name('image semantic'), contribution(0.009) ),
  term( name(organizing), contribution(0.022) ),
  term( name(learning), contribution(0.001) )
]
```

- to make contextual information, such as the URL of a reviewer's DBLP page, conveniently available to web applications without them incurring the cost of additional web service queries to fetch this information separately.

Therefore, in practice, the `match` predicate will include more that just the similarity score we have shown in our earlier examples.

When we abstracted from the SubSift prototype web application's structure to the `subsift/3` predicate definition we ignored some important practical details. We now return to examine these and review whether they can be addressed by virtue of the overall workflow and component-based approach, for example whether their function can use existing web services elsewhere, or whether they need to be incorporated into the generalised submission sifting workflow.

- For reviewers, the `document/2` predicate requires some textual representation of each reviewer's DBLP author page – either the entire HTML page or some subpart of it that includes just the publication history. In the SubSift prototype this functionality was implemented as the server-side part of an interactive web form. From a workflow perspective, there is no reason why this functionality needs to be incorporated into the generalised submission sifting workflow as it is specific to one use case. For this reason we will assume that fetching publication data from DBLP is implemented in some separate predicate, the results of which are supplied to `subsift/3` in the `Reviewers` parameter.
- Discretisation of the similarity scores into bids depends on the availability of bid thresholds. There either needs to be a way to pass the bid thresholds through to `match/3` or else thresholding needs to be performed as a post-processing step on the `Matches` output variable. Although bids only relate to one of our use cases, and so could arguably be handled later in a separate predicate, for convenience and efficiency we chose to pass in threshold parameters and handling discretisation within `subsift/3`.

- There are various options relating to the transformation of text into a bag-of-words representation that were hard-coded into the SubSift prototype. Some examples include the following.
  - Ignoring case – so that `"Cartesian"` ⇔ `"cartesian"`.
  - Ignoring punctuation – so that `"very very big"` ⇔ `"very, very big!"`.
  - Removing *stop words* (terms known to occur frequently in the language) – so that `"a very very big dataset"` ⇔ `"big dataset"`, assuming we define `a` and `very` to be stop words.
  - Synonym substitution – so that `"data set"` ⇔ `"data-set"`.
  - Including n-grams (sequences of n words treated as a single term) for $n \subseteq \{1,2,3,4,5\}$ – so that for $n = \{2,4\}$, `"a big data set"` maps to the set of terms, { `a_big`, `big_data`, `data_set`, `a_big_data_set` }.

  To process the different types of textual input (e.g. plain text, web pages, blog posts) involved in our use cases, it is necessary to pass these options to the `profile/2` predicate.

Consequently, to complete our generalised submission sifting structure we must introduce a way to pass parameters into `subsift` and on to the predicates in the body of the definition. We do this through the addition of a *context* parameter, `C`, to the signature of `subsift`, resulting in the following signature of `subsift/4`.

Definition 3.1.3 (Generalised Submission Sifting Signature) *Generalised submission sifting, `subsift/4`, is a predicate with signature,*

```
subsift(+Reviewers, +Papers, +C, -Matches).
```

The context parameter `C` is a key-value association list where each key is a unary functor and each value is its argument, enabling the passing of multiple parameters, e.g. `C = context(bid3(0.64), bid2(0.05], bid1(0.01), ignorecase(true)).` Incorporating this parameter into the body of the `subsift/4` predicate, we arrive at our definition of generalised submission sifting.

Definition 3.1.4 (Generalised Submission Sifting) Generalised Submission Sifting *is a predicate `subsift/4` with the definition,*

```
subsift(+Reviewers, +Papers, +C, -Matches) :-
    document(Reviewers, C, D1),
    document(Papers, C, D2),
    profile(D1, C, P1),
    profile(D2, C, P2),
    match(P1, P2, C, Matches).
```

where input variables `Reviewers` and `Papers` are unified with the set of representations of reviewers and set of representations of papers respectively. Input variable `C` (context) is unified with a higher-order structure specifying parameters to the pairwise comparison. Output variable `Match` is unified with a list of reviewer-paper pairs ranked by descending similarity.

To translate Definition 3.1.4 into a workflow, we ignore the strictly sequential procedural interpretation of Prolog clauses in the body of `subsift/4` and instead trace the dependencies back from output variable `Matches` to the input variables `Reviewers` and `Papers`. This results in the workflow graph structure depicted in Figure 3.4. Each body predicate, `document`, `profile` and `match`, corresponds to a web service component in the generalised submission sifting workflow, which we define more formally below.

Definition 3.1.5 (Generalised Submission Sifting Workflow) Generalised Submission Sifting Workflow *is the graph induced by resolution of following program,*

```
profile1(+Reviewers, +C, -Profiles) :-
    document(Reviewers, C, D1),
    profile(D1, C, Profiles).

profile2(+Papers, +C, -Profiles) :-
    document(Papers, C, D2),
    profile(D2, C, Profiles).

subsift(+Reviewers, +Papers, +C, -Matches) :-
    profile1(Reviewers, C, P1),
    profile2(Papers, C, P2),
    match(P1, P2, C, Matches).
```

in response to the query,

```
?- subsift(Reviewers, Papers, C, Matches).
```

where input variables, `Reviewers`, `Papers`, `C`, *and output variable,* `Matches`, *are bound as stated in Definition 3.1.4 and predicates,* `document/3`, `profile/3` *and* `match/4`, *are defined as web services.*

Having finally arrived at our definition of a generic workflow, we next move on to describe Subsift Web Services, our implementation of this workflow, through which we demonstrate the applicability of an workflow and component-based approach to addressing our motivating use cases.

3.2 SubSift Web Services

SubSift Web Services, which from hereon we will refer to as just SubSift, are hosted as a freely available resource by the University of Bristol. Further details can be found on the SubSift website[7] along with extensive documentation and a number of demonstrations. The SubSift software is published under an open source licence and is available from the `subsift` subversion repository on Google Code[8].

Figure 3.5 shows the high-level design of the SubSift system, with its REST web services API and supporting web harvester robot. Details of the API itself follow

[7] SubSift website – `http://subsift.ilrt.bris.ac.uk`, visited May 2014.

[8] SubSift source code – `http://code.google.com/p/subsift/`, visited May 2014.

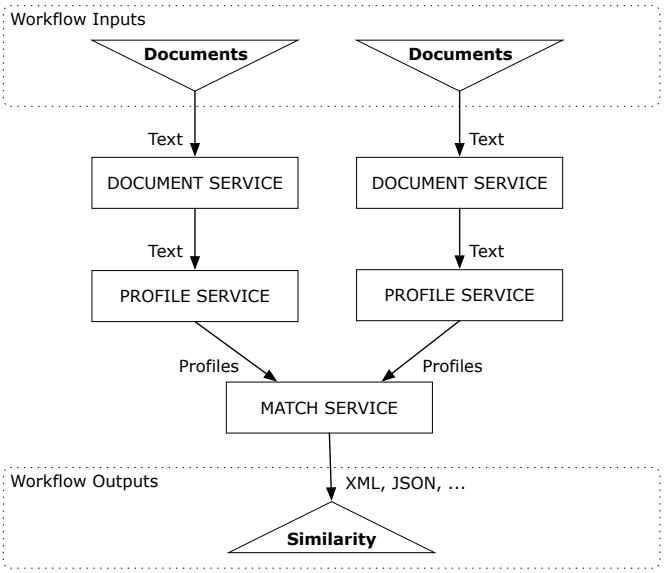


Figure 3.4: Generalised Submission Sifting Workflow.

immediately below and further details appear throughout the workflow use case descriptions in the next section.

However, before moving on to describe the API it should be emphasised that apart from the initial data ingest and eventual download of matching results, only metadata identifying documents in the filestore is transmitted at each web service call; expensive HTTP transmission of document data is not required between processing steps because all the SubSift web services have access to the same local filestore. In other words, the local and public services are not conflated as might at first appear from this and subsequent workflow schematics in this chapter. Furthermore, although out implementation of SubSift does not extend to large datasets, there is no reason why the architecture could not support mapped local filestore and use highly efficient non-HTTP data transfer to move data into and out of the filestore. For the use cases discussed in this Thesis, HTTP transmission is sufficient – not least because the dependancy on an inherently HTTP-based web harvester rate limits ingest from academic homepages or online bibliographies.

The SubSift REST API is organised around a series of *folders* into which data *items* are stored. This organisation is modelled on the familiar filing system concept of folders and files. The three main folder types are documents, profiles and matches. The first request below would create a documents folder called `staff` and the second would list its items[9].

```
curl -X POST <uri>/<user_id>/documents/staff
curl -X GET <uri>/<user_id>/documents/staff/items
```

In SubSift, a document is a piece of text to be profiled and matched. A document

[9]We omit the security token needed in the request header of `DELETE`, `POST` and `PUT` requests, and of all requests for folders marked as private.

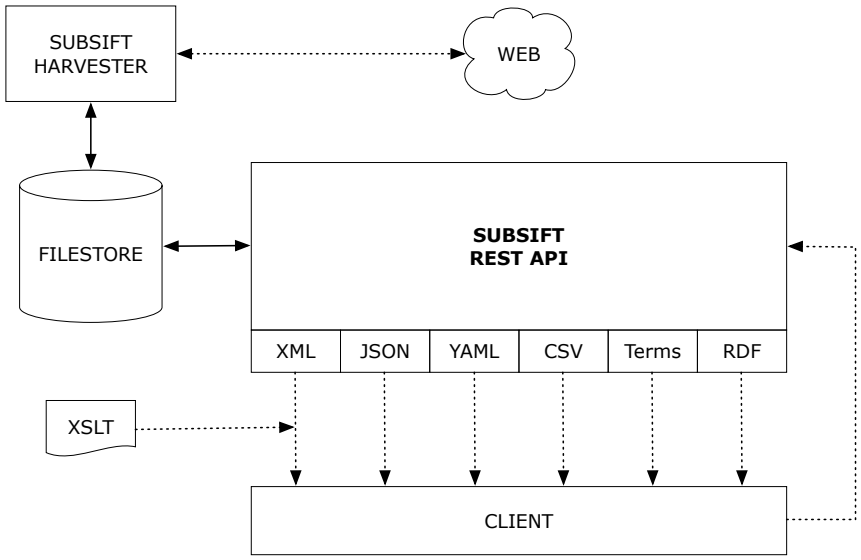


Figure 3.5: SubSift System Architecture. *Dotted lines represent HTTP. Solid lines represent file access. XML, JSON, YAML, CSV, Terms and RDF are response formats. Terms are Prolog terms. XSLT stylesheets are invoked via optional inclusion of XSL processing command in XML returned by SubSift.*

will usually be the text from some external source such as the text of a web page or a conference paper abstract. A profile in this context is a summary representation of the features of a single document, with respect to the other documents in the same documents folder. A profile is produced by some profiling function (see Definition 2.1.5). In the example fragment of a profile item below, the person's names are the most discriminating terms, occurring 77 times in the source DBLP author webpage[10] and having a tf-idf of 0.404 (last name) and 0.246 (first name) relative to the rest of the items in this profiles folder. The terms *logic* and *ILP* also rank highly.

```
profile(
    id('Peter Flach'),
    source('http://dblp.../Flach:Peter_A=.html'),
    term( name(flach),
        n(77),
        idf(6.047),
        tf(0.067),
        tfidf(0.404)),
    term( name(peter),
        n(77),
        idf(3.685),
        tf(0.067),
        tfidf(0.246)),
    term( name(logic),
        n(18),
        idf(3.924),
```

[10]DBLP Computer Science Bibliography – `http://dblp.uni-trier.de/`, visited April 2014.

```
        tf(0.016),
        tfidf(0.061)),
    term( name(ilp),
        n(14),
        idf(4.685),
        tf(0.012),
        tfidf(0.057)),
```

One usage of profiles in SubSift is to obtain a list of distinguishing terms, or keywords, for a document – for example, automatically extracting keywords from abstracts of papers submitted to a conference. Another usage is for two profile folders to be compared against each other to produce a matches folder. A matches folder is produced by some matching function (see Definition 2.1.6) and is thus created in this context by analysing every pairing of profile items drawn from the two profiles folders. Each match item records the tf-idf cosine similarity, and various related statistics, of a single profile from the first profiles folder against every profile from the second profiles folder. A typical usage of such a comparison is to match submitted conference paper abstracts with the bibliography pages of Programme Committee (PC) members (i.e. reviewers) in order to rank potential reviewers for each paper and vice versa, as depicted in Figure 3.6.

SubSift's API methods make intermediate data and metadata available for all folders and items. For example, the entire similarity matrix of a match folder can be exported or the relative contribution of each term towards a particular match item's similarity can be retrieved. In the match item fragment below, metadata about the similarity score of 0.076, between a person and a paper, shows that the terms *logic* and *ILP* made large contributions to the score relative to other terms.

```
match(
    id('Peter Flach'),
    source('http://dblp.../Flach:Peter_A=.html'),
    item( name('paper id removed'),
        description('paper title removed'),
        score(0.076),
        source(text),
        term( name(logic), contribution(0.005) ),
        term( name(ilp), contribution(0.004) ),
        term( name(class), contribution(0.001) ),
        term( name(inductive), contribution(0.001) ),
```

For ease of integration, there is flexibility in both input and output. Document text may be added per item or in bulk or by supplying a list of URLs to be fetched asynchronously by SubSift's harvester robot. The API methods can return data in the following representational formats: CSV, JSON, RDF, XML, YAML, and Prolog terms.

3.2.1 Example REST API Enactment of Submission Sifting

In this section we give a flavour of how the workflows in this chapter can be enacted through sequences of REST method calls to SubSift's web services. As our example

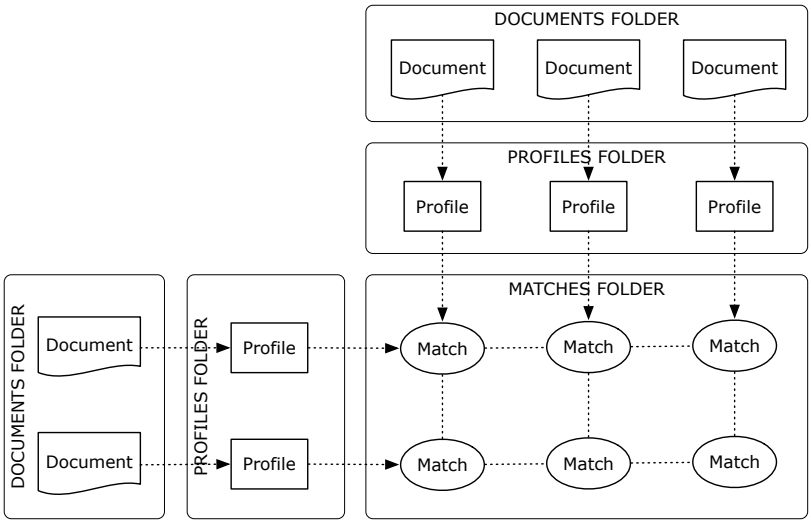


Figure 3.6: Generalised submission sifting enacted using the SubSift REST API. *A sequence of API method calls transform a pair of document folders (e.g. abstracts and reviewers) into a folder of match statistics. Original and computed data plus metadata can be obtained at each step via API methods.*

we choose the generalised submission sifting workflow (Definition 3.1.5). In steps 1–5, profiles of conference PC members are created; in steps 6–8, profiles of the submitted abstracts are created; and in steps 9–10, the two sets of profiles are pairwise matched against each other and the results published to the web. For readability, at each step the HTTP request methods and their parameters are denoted using the format below and no response values are shown.

```
<http_method> [<uri>][<user_id>]<path>
<parameter_name_1> = <parameter_value_1>
<parameter_name_2> = <parameter_value_2>
...
<parameter_name_N> = <parameter_value_N>
```

For brevity in the following examples we will omit `<uri>`, which has the value `https://subsift.ilrt.bris.ac.uk` for the publicly hosted version of SubSift, and `<user_id>` which will always be the account name (e.g. `ecmlpkdd12` for the ECML-PKDD'12 conference). Note also that we omit details of the security token needed in the HTTP request header of all `DELETE`, `POST` and `PUT` requests. The token is also required to access folders and data marked as private, irrespective of request method.

Step 1. Obtain a list of PC member names and their DBLP author page URIs. SubSift's DBLP Author Finder demo accepts a list of author names and then looks up these names on the DBLP Computer Science Bibliography and suggests author pages which, after disambiguation, are returned as a list with each line as:

```
<pc member name>, <uri>
```

The SubSift API has a type of folder, called a bookmarks folder, specifically for defining lists of URIs. So, to begin our example we first create a bookmarks folder to hold the list of PC member URIs.

```
POST /bookmarks/pc
```

Step 2. Create bookmarks in this folder, one per PC member URI.

```
POST /bookmarks/pc/items
items_list=<list of URIs from step 1>
```

Step 3. Create a documents folder to hold the web page content (text) of the DBLP author pages.

```
POST /documents/pc
```

Step 4. Import the bookmarks folder into the documents folder. This adds the URIs to SubSift Harvester Robot's crawl queue. We name the documents folder the same as the bookmarks folder. This is a convention, not a requirement, but makes the ancestry of the folder obvious.

```
POST /documents/pc/import/pc
```

In time, all the URIs will be fetched and a document created in the documents folder for each webpage fetched. To detect if there are still URIs waiting to be fetched, applications may poll the same URI until an HTTP 404 error is returned.

```
HEAD /documents/pc/import/pc
```

Step 5. Create a profiles folder from the bookmarks folder.

```
POST /profiles/pc/from/pc
```

Step 6. For bulk upload, pre-process the abstracts into CSV format so that each line is: `<paper id>, <abstract>`. Include the text of the paper title in with the abstract text. Create a documents folder to hold the abstracts.

```
POST /documents/abstracts
```

Step 7. Use the abstracts CSV text to create a document item for each abstract.

```
POST /documents/abstracts/items
items_list=<csv from Step 6>
```

Step 8. Create a profiles folder from the documents folder.

```
POST /profiles/abstracts/from/abstracts
```

Step 9. Match the PC members profiles folder against the abstracts profiles folder.

```
POST /matches/pc_abstracts/profiles/pc/with/abstracts
```

Having generated the matches data it may then be retrieved in a variety of ways. For example, to fetch the ranked list of papers per PC member and then to fetch the ranked list of reviewers per paper and finally retrieve the similarity matrix to use for bidding, optionally specifying manually chosen thresholds to discretize the scores into the range 3..1 as bid values.

```
GET /matches/pc_abstracts/items
profiles_id=pc

GET /matches/pc_abstracts/items
profiles_id=abstracts

GET /matches/pc_abstracts/matrix
```

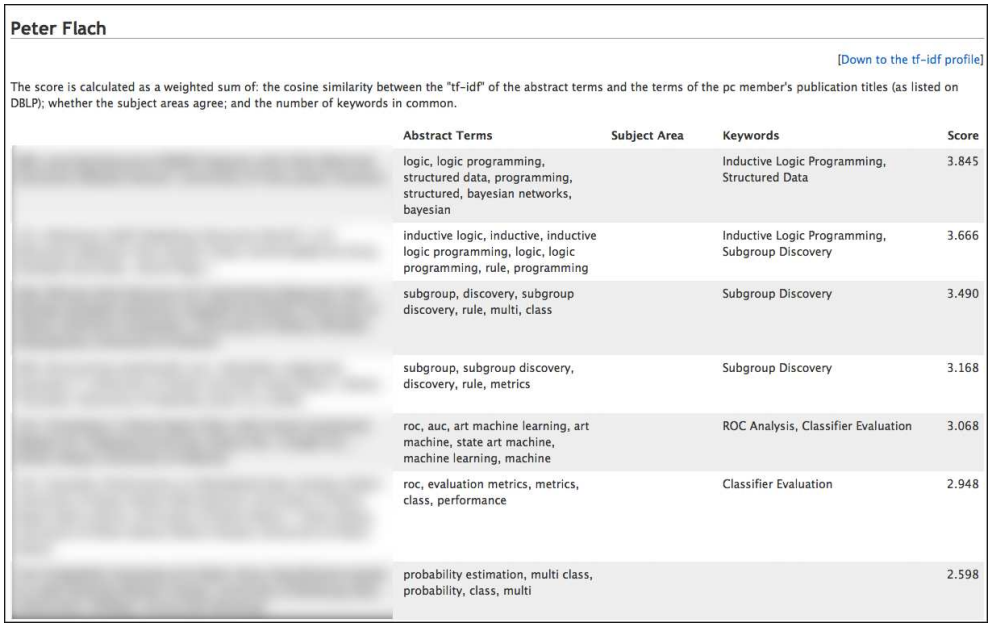
Step 10. Generate and publish profiles and matches as human-readable reports to the web or download them as a zip of webpages to publish on the user's own site.

```
POST /reports/pc/profiles/pc
POST /reports/abstracts/profiles/abstracts
POST /reports/pc_abstracts/matches/pc_abstracts
```

The published reports appear on the web at `/reports/pc`, `/reports/abstract` and `/reports/pc_abstracts` and are either privately or publicly accessible. Examples of two parts of an individualised report generated from ECML-PKDD'12 are shown in Figures 3.7 and 3.8.

Although not immediately obvious from the examples in this section, SubSift's URI design and RDF support automatically connect its shared data to the Linked Data graph and wider Semantic Web. In most of the API examples given, simply appending the suffix `.rdf` to the URL will return data in RDF format, with absolute URI references to resources such as bookmarks and documents. We make a note of this design feature here as it is something that we will return to in the future work chapter of this Thesis.

Having described the generalisation of SubSift from one specific workflow into a family of web services capable of enacting generalised submission sifting, we now turn to research question of whether this generic profiling and matching paradigm can be applied in other research intelligence settings.



Peter Flach

[Down to the tf-idf profile]

The score is calculated as a weighted sum of: the cosine similarity between the "tf-idf" of the abstract terms and the terms of the pc member's publication titles (as listed on DBLP); whether the subject areas agree; and the number of keywords in common.

	Abstract Terms	Subject Area	Keywords	Score
[redacted]	logic, logic programming, structured data, programming, structured, bayesian networks, bayesian		Inductive Logic Programming, Structured Data	3.845
[redacted]	inductive logic, inductive, inductive logic programming, logic, logic programming, rule, programming		Inductive Logic Programming, Subgroup Discovery	3.666
[redacted]	subgroup, discovery, subgroup discovery, rule, multi, class		Subgroup Discovery	3.490
[redacted]	subgroup, subgroup discovery, discovery, rule, metrics		Subgroup Discovery	3.168
[redacted]	roc, auc, art machine learning, art machine, state art machine, machine learning, machine		ROC Analysis, Classifier Evaluation	3.068
[redacted]	roc, evaluation metrics, metrics, class, performance		Classifier Evaluation	2.948
[redacted]	probability estimation, multi class, probability, class, multi			2.598

Figure 3.7: Top section of a SubSift personalised report listing top ranked papers. *Titles and authors of the papers have been blurred out to preserve confidentiality.*

Profile: Peter Flach

The following is a list of terms ordered by "tf-idf" score. The higher the score, the more discriminating the term is with respect to the other profiles. The list shows the term itself (i.e. keyword), the term count (#, i.e. how many times the term occurs within the profiled text), term frequency (tf, i.e. how frequent the term is within this text), inverse document frequency (idf, i.e. how infrequent the term is across all the texts profiled), and "tfidf" (i.e. the product of tf and idf which scores most highly terms which are most discriminating for this text with respect to the other texts).

Term	#	tf	idf	tf*idf
roc	14	0.007	4.915	0.036
inductive	14	0.007	3.429	0.025
roc analysis	7	0.004	6.388	0.024
logic	12	0.006	3.255	0.021
subgroup discovery	7	0.004	5.310	0.020
subgroup	7	0.004	5.310	0.020
inductive logic	8	0.004	4.558	0.019
structured data	7	0.004	4.604	0.017
inductive logic programming	7	0.004	4.558	0.017
order	12	0.006	2.460	0.016
logic programming	7	0.004	4.201	0.015

Figure 3.8: Bottom section of a personalised report showing PC member profile.

3.3 Profiling and Matching Workflow Demonstrations

In this section we demonstrate the versatility of the generalised submission sifting workflow from Definition 3.1.5 by showing how, with minor modifications, similar profiling and matching workflows may be defined to address each of our motivating use cases from Chapter 1. We also demonstrate the flexibility our SubSift framework through proof of concept implementations of each use case as a web application or workflow. We also describe our implementation of the "Related Use Cases" from Chapter 1, which further demonstrates the utility of the SubSift framework for

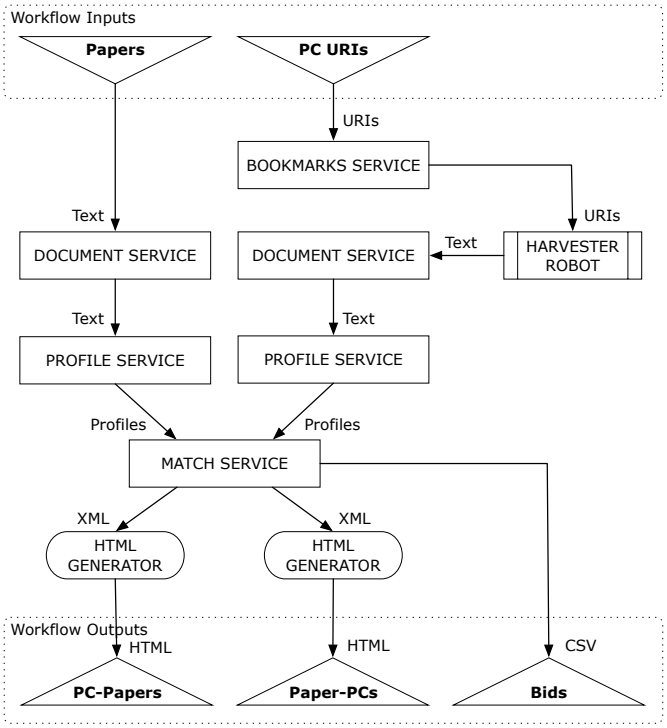


Figure 3.9: Schematic of *submission sifting* workflow.

analysing general textual content in a web application context, but also uncovers a number of limitations that provide motivation for work detailed in the next chapter.

3.3.1 Use Case 1 – Submission Sifting

A SubSift demonstrator based on this workflow was implemented as a wizard-like series of web forms, taking the PC chair through the above process form by form[11]. On the first form, a list of PC member names is entered. SubSift looks up these names on DBLP and suggests author pages which, after any required disambiguation, are used as documents to profile the PC members. Then the conference paper abstracts are uploaded as a CSV file and their text is used to profile the papers. After matching PC member profiles against paper profiles, SubSift produces reports with ranked lists of papers per reviewer, and ranked lists of reviewers per paper. Optionally, by manually specifying threshold similarity scores or by specifying absolute quantities, a CSV file can be downloaded with initial bid assignments for upload into the conference management tool.

The schematic for this workflow are depicted in Figure 3.9. This is almost the generic workflow from Figure 3.4, apart from the inclusion of the Bookmarks Service to manage the list of URIs and SubSift's Harvester Robot to fetch the web pages. The HTML Generators use XSLT to transform XML into HTML reports.

A Web 2.0 implementation of this workflow has been released on the SubSift website as a publicly available demonstration and as a practical tool for conference

[11] Sift demo – `http://subsift.ilrt.bris.ac.uk/demo/sift`, visited April 2014.

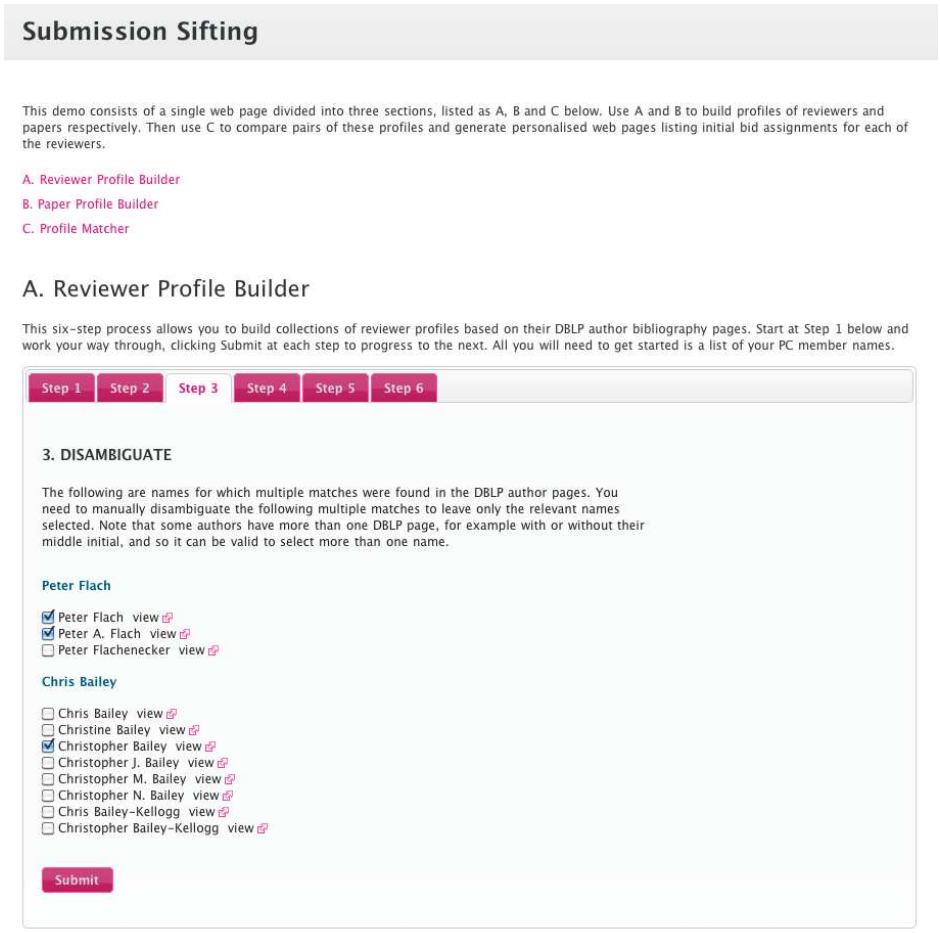


Figure 3.10: Web 2.0 wizard implementation of the *submission sifting* workflow, showing the Disambiguation step of the Reviewer Profile Builder. In the example, Peter Flach and Peter A. Flach are both selected because both of the corresponding DBLP pages refer to the same author.

organisers, in this example, from the domain of computer science. The earlier SubSift demonstrator did not make use of web services and was instead a single self-contained web application. The new submission sifting demonstration is an entirely client-side JavaScript workflow enactment which integrates calls to the SubSift web services with calls to additional web services for disambiguating DBLP author homepages, extracting publication details from DBLP author pages and for aggregating multiple web pages into a single bag-of-words. An example of the wizard-style dialogue presenting the user with disambiguation choices is shown in Figure 3.10.

3.3.2 Use Case 2 – Finding an Expert

In this workflow the user submits a fragment of text, typically an abstract or the full text of a paper, and SubSift compares this against the publications of a pre-defined list of researchers. The result is a list of researchers ranked by their similarity to the submitted fragment. Figure 3.11 shows *ILRT Matcher*, an entirely client-side,

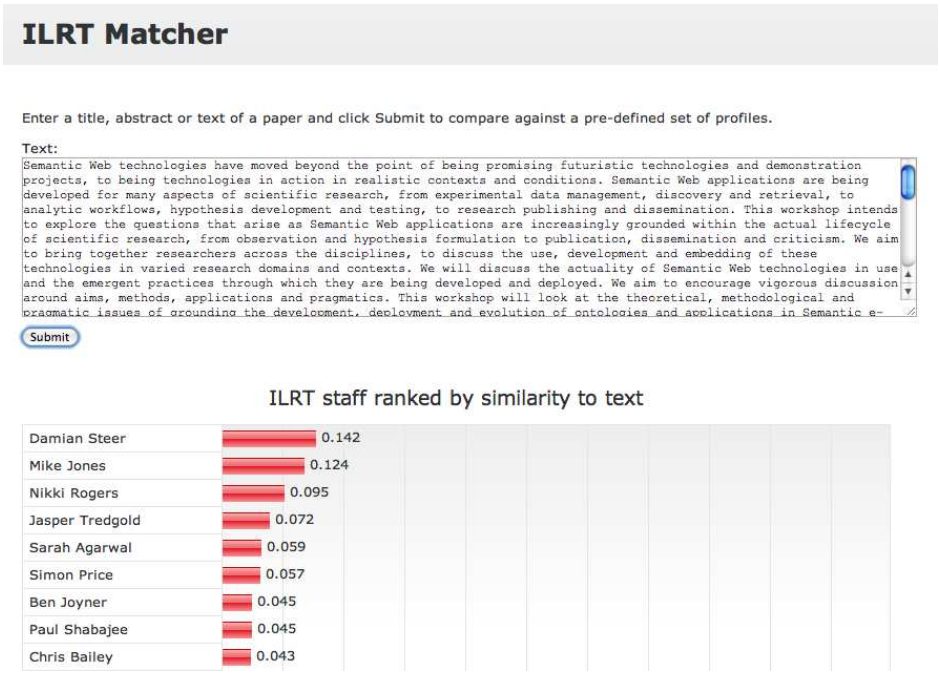


Figure 3.11: Similarity match between text fragment and ILRT staff.

Web 2.0 JavaScript implementation of this workflow, comparing the submitted text against every ILRT staff member's homepage[12]. The results are displayed as a bar chart. Optionally, the list of terms contributing to each researcher's similarity score can be viewed along with the percentage that each term contributed to the combined score, as shown in Figure 3.12.

This two-phase workflow is depicted in Figure 3.13. The first phase is preparatory and creates all the necessary folders and items for the second, interactive phase. The second phase is enacted each time a user submits text via the Web 2.0 form. In the first phase a null (empty) text fragment is used to form a singleton item in a documents folder to be profiled and compared against the staff web page profiles.

Bespoke versions of this demonstrator are currently used as a recommender system by editors of two leading computer science journals: *Machine Learning* and *Data Mining and Knowledge Discovery*.

3.3.3 Use Case 3 – Visualising Similarity Networks

To visualise the similarity within the ILRT group at the University of Bristol, we used the XML output from the SubSift match service as input to an XSLT stylesheet that transformed the similarity scores into edges on a graph. Tools like Graphviz[13] can be used to visualise a textual representation (DOT format) of the graph as shown in Figure 3.14 for the homepages of ILRT staff members.

This workflow in Figure 3.15 compares a profiles folder against itself to rank

[12]ILRT Matcher – `http://subsift.ilrt.bris.ac.uk/demo/ilrt_matcher`, visited April 2014.

[13]Graphviz – `http://www.graphviz.org`, visited April 2014.

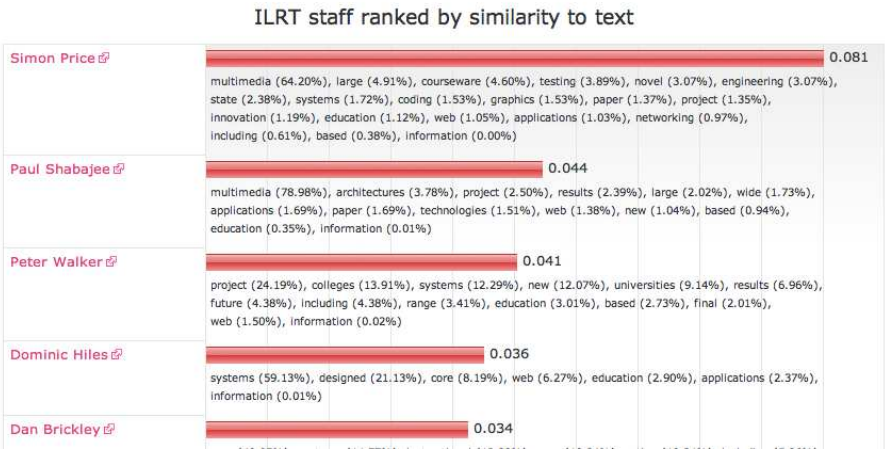


Figure 3.12: Term contributions for matches between text and ILRT staff.

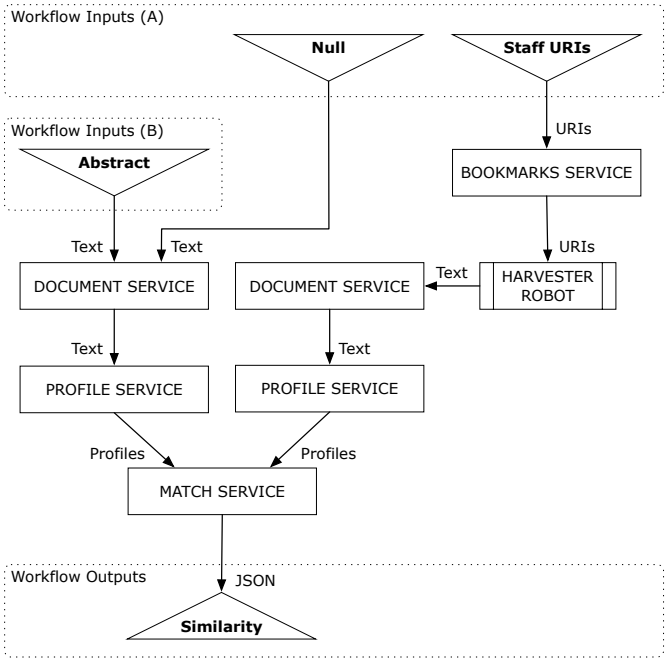


Figure 3.13: Schematic of *Finding an Expert* workflow.

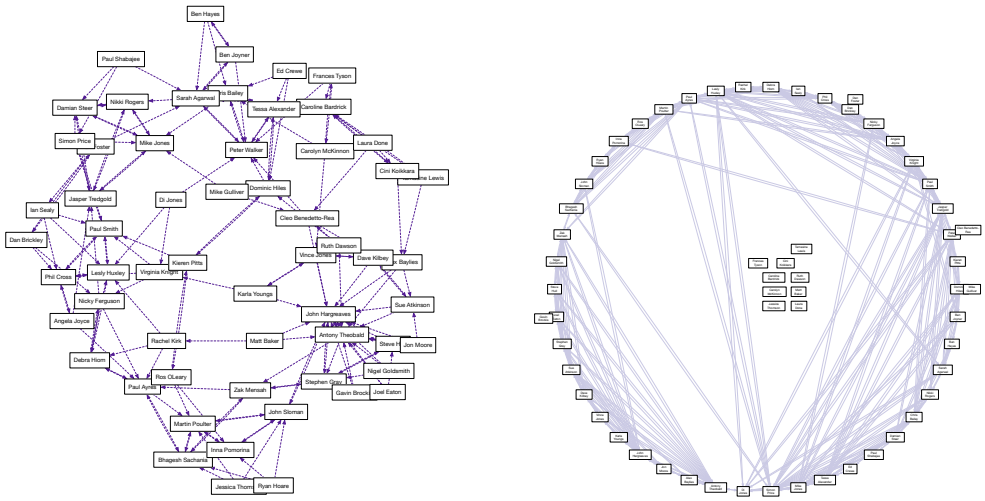


Figure 3.14: ILRT staff homepage similarity matches data output to DOT format and rendered in Graphviz (left: force-directed layout, right: circular).

all the documents profiled by their similarity to each other. Ignoring the reflexive matches (a profile is always identical to itself), the resultant similarity data returned by SubSift is then transformed by a SubSift XSLT file from XML into a textual DOT file suitable for rendering as an image in Graphviz.

3.3.4 Use Case 4 – Profiling Reading Lists

As proof of concept, the text of web pages linked to social bookmarks files of a member of the Intelligent Systems Laboratory was used to create a profiles of a researcher's interests. Such information is often displayed on homepages as a tag cloud, such as Figure 3.16 which was produced in Wordle[14] using SubSift profile terms. This information may also be published as Semantic Web data using formats like Friend of a Friend (FOAF).

A SubSift workflow to achieve this is shown in Figure 3.17. The bookmarks are harvested to create the researcher's web reading list profile. The researcher's bookmarks are not being compared to those of other researchers here and so the ranked list of terms produced will contain high scoring terms, such as the person's name, that are not relevant. To filter these out, the terms of the ACM Computing Classification System (CCS)[15] are used during profiling to ensure that all terms output from the Profile Service occur in the CCS list.

3.3.5 Use Case 5 – Ranking News Stories

Figure 3.18 shows a two-phase workflow based on an extension to the submission sifting workflow that can be used to reorder the stories in an RSS file according to their similarity to the researcher as defined by their own published works. In the first phase, the current homepage of the researcher is used to create their profile. This

[14] Wordle – `http://wordle.net`, visited April 2014.
[15] ACM CCS – `http://www.acm.org/about/class/1998/`, visited April 2014.

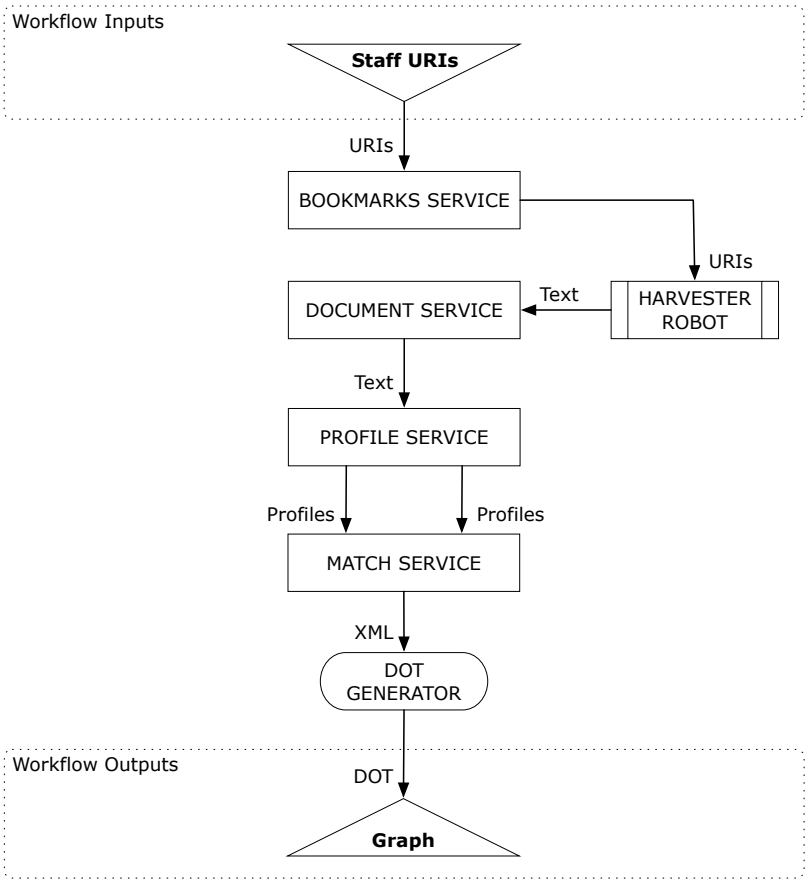


Figure 3.15: Schematic of *Visualising Similarity* workflow.

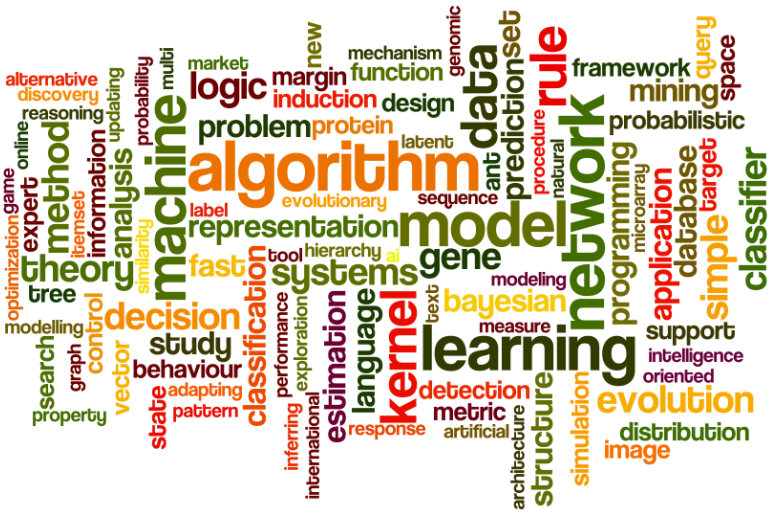


Figure 3.16: Tag cloud visualisation of researcher interests.

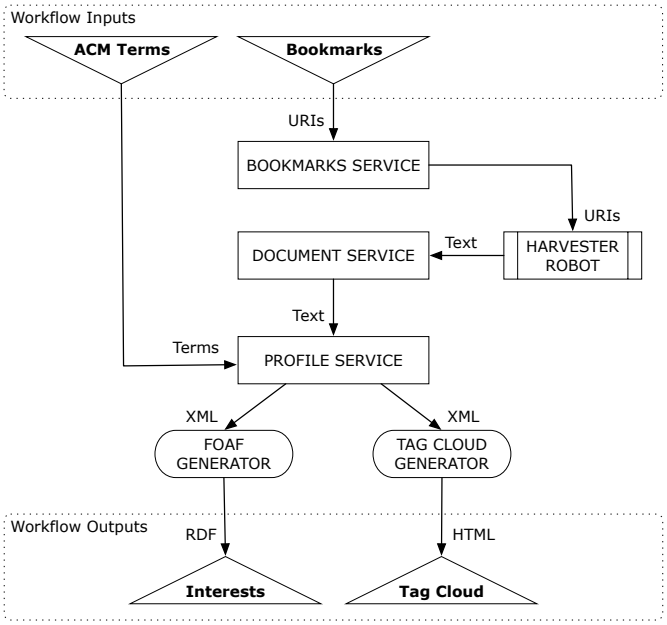


Figure 3.17: Schematic of *Profiling Reading Lists* workflow.

profile only needs recalculating periodically. In the second phase, an RSS feed is used as a list of URIs which are harvested and used to create a profile for each story. The story profiles are matched against the reviewer profile and the resultant ranking is used to reorder the original RSS file.

An interesting characteristic of this workflow is the separation of flow from the steps for reconfiguration. Clearly there are synchronisation risks in dividing this workflow into two phases that run at separate times: it is not possible to rank stories in the RSS file during phase B if the profiling of the corresponding user homepage has not been previously completed during phase A. Careful checking for the existence of profiles before attempting the match can guard against this but what is less obvious is the importance of the word '*corresponding*' in the preceding sentence. For this workflow to be useful to a user their homepage profiles generated in phase A must persist and be correctly identified when the user later triggers phase B. In practice, this either adds a requirement that users are uniquely identified through some login process or that they each have their own SubSift account. Neither of these requirements is zero cost and this potentially places this particular use-case in a different class to all the others presented here; in the other use cases, implementation is possible such that user identity is only peristed for the duration of a browser session. In our proof of concept implementation of this workflow we assumed one user per subsift account, in effect delegating the identification of the user to the SubSift API.

The above use case workflows and demonstrators implemented in the SubSift framework demonstrate the flexibility of this e-Science inspired approach for analysing general textual content in a web application context. In the next section, we move beyond our original motivating use cases to investigate other applications of the profile-

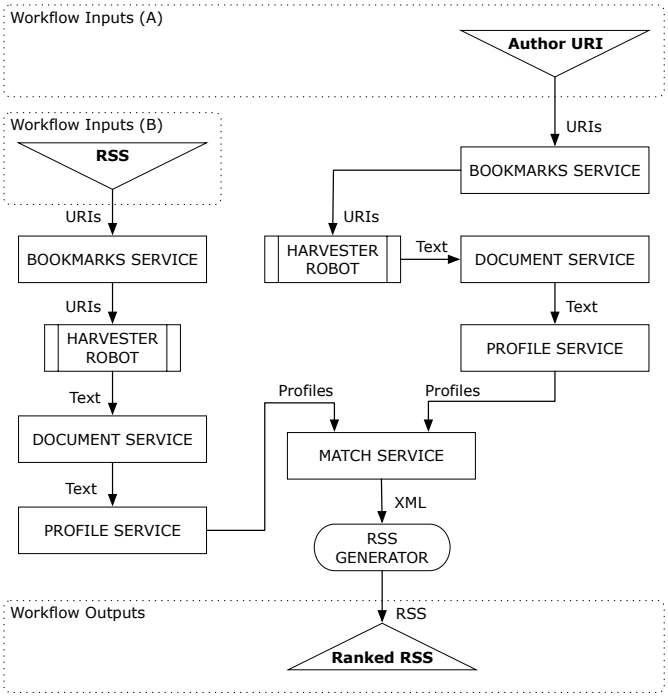


Figure 3.18: Schematic of *Ranking News Stories* workflow.

match paradigm in the research intelligence domain.

3.3.6 Related Use Cases

In Section "1.1.6 – Related Use Cases" of Chapter 1 we outlined four use cases for mining and mapping the research landscape of the University of Bristol. This work, part of a pilot project called *ExaMiner* [PF13b], aimed to provide the University's researchers and Research and Enterprise Development (RED) group with new ways of searching, accessing and visualising connections between existing research, irrespective of the organisational boundaries and structures. By exploiting the flexibility of the workflow and component-based approach we were able to successfully implement web application demonstrators for all four of these ExaMiner use cases – mainly in client-side JavaScript, using SubSift's existing web services and variations of the workflows already described. Below we briefly describe workflows implementing these ExaMiner use cases to further explore the flexibility of SubSift model.

- **Find a Researcher**. The user pastes some text into a web form, for example a few keywords or the abstract of a paper, and the application finds the most similar researchers in a selected department. This is a scaled-up example of "Use Case 2 – Finding an Expert", which we previously demonstrated for a single department, conference programme committee or journal editorial board. Considering the increased number of researchers, to keep computation within acceptable user response times and to avoid web server process timeouts, our workflow design used a separate set of profiles for each department but with the addition of caching it would be possible to aggregate all

researchers' profiles into a single University-wide set of profiles. In Chapter 4 we introduce another framework implementation, developed subsequent to the ExaMiner project, that is more scalable than SubSift and so could have offered an alternative to bespoke caching here.

- **Find Similar Research**. The user defines the research they would like to find by browsing through and selecting examples from a department-grouped list of all the University's researchers. The list of selected researchers' homepages are all concatenated into a single document which is then profiled and matched against the researchers from a department chosen by the user (Figure 3.19). In other words, this is another variation of "Use Case 2 – Finding an Expert", but instead of pasting some text into a form, the text to be matched is constructed behind the scenes from the text of all the selected researchers' homepages. To implement this efficiently requires a new web service in SubSift to avoid the need for transmission of homepage text back and forth between the client and server, something that would otherwise be necessary to concatenate every selected homepage's text into a single string prior to profiling.

- **Find Research Networks**. The user selects one or more departments and the application cross-matches all their researchers to find networks of similar research, displaying them as a similarity network diagram. This workflow is almost the same as the "Use Case 1 – Submission Sifting", using the same predefined sets of researchers used in the above two use cases but using the same presentation mechanism as "Use Case 3 – Visualising Similarity Networks". In the original requirements for this workflow, comparisons were to have been made possible between larger organisational units (e.g. schools or faculties) and, potentially, between institutions. However, the memory requirements and elapsed time for computing pairwise similarity in this use case both grow quadratically, $O(n^2)$, where n is the mean number of researchers in the organisational units being compared. The order of magnitude increase in the number of profiles to generate and cross-match exceeds the current capacity of SubSift and/or typical time-out limits of web servers. Recall that the context of this Thesis is web applications. Hence, SubSift was designed with this web application context in mind and so uses in-memory calculation to give rapid results for pairs of datasets of, typically, only a few hundred records. In Chapter 4 we introduce an alternative computational model that potentially overcomes this limitation. However, with the addition of extensive pre-computation and caching it would also be possible to achieve acceptable responses in SubSift.

- **Profile Email Recipients**. The user pastes a list of University of Bristol email addresses into a web form and the application displays each person's full name, job title, contact details, homepage url and research profiles, optionally also matching them to find networks of similar research (Figure 3.20). In the previous three use cases researchers were profiled based on their staff homepages using an approach that could, in principle, be applied to any public-facing organisational website. However, this fourth use case tries out a different approach by implementing a bespoke web service to enable querying of the University's corporate databases, returning staff details and publication records to the web

Figure 3.19: *Find Similar Research* example: A combined publications profile is produced for a selection of people in the Department of Drama, Theatre, Film and Television and matched against individual profiles of staff in the Department of Archaeology and Anthropology to produce the similarity chart.

application. The application then submits these details to SubSift to dynamically create a set of researchers to profile and cross-match in the same way as "Use Case 3 – Visualising Similarity Networks". This demonstrator was used to support the Bristol Biomedical Research Unit in Nutrition workshop, organised by RED in September 2012. RED had conducted a pre-workshop survey of participants' research interests and their expectations for the event and this was merged into the staff profile information by pre-processing the data.

That these four ExaMiner use cases could be implemented as SubSift workflows in the context of an enterprise-wide project further demonstrates the flexibility of the framework and the general workflow and component-based approach. Although we have no comparative data for implementing ExaMiner without the use of SubSift, we suspect that it would not have been feasible within the limited pilot budget and tight timescales involved.

3.4 Summary

Submission sifting is the problem of matching submitted conference or journal papers to potential peer reviewers based on the similarity between the paper's abstract and the reviewer's publications as found in online bibliographic databases. This application of the vector space model from information retrieval to the submission sifting problem was demonstrated to produce useful results in practice. It was also shown that the problem can be decomposed into separately re-usable components of

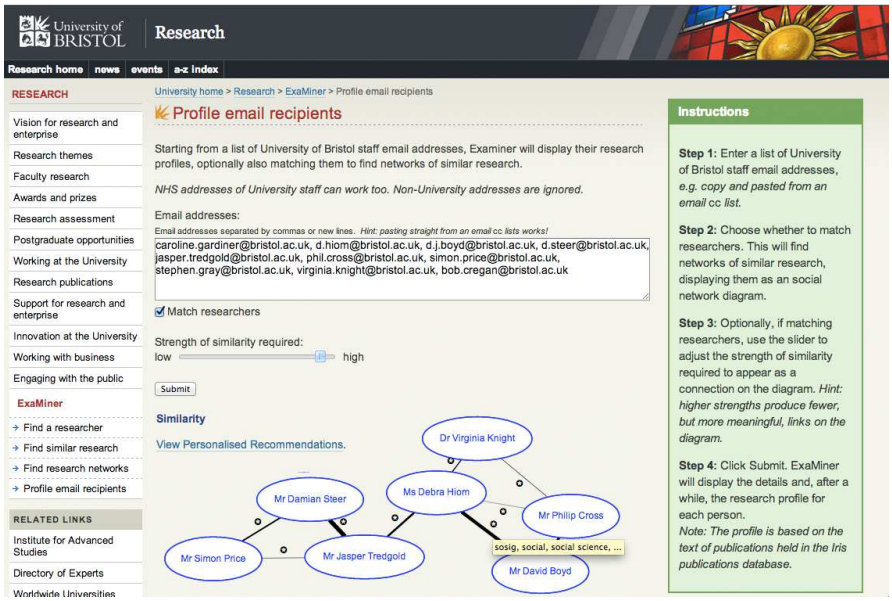


Figure 3.20: *Profile Email Recipients* example: Email addresses pasted straight from an email 'to' or 'cc' list are used to look-up staff details and publications from corporate databases and produce profiles which are then cross matched. Similarities above the threshold set on the slider are shown as links on the graph. Line thickness is proportional to similarity. Mouse rollover on lines shows top matching terms.

a generic submission sifting workflow. The software, known as SubSift, has been used to support several major machine learning and data mining conferences.

The generic submission sifting workflow and its components were abstracted to define a general framework to enable web services to be assembled into workflows to analyse heterogeneous textual content ranging from documents and web sites. A range of workflows were used to investigate the utility of this framework in creating applications to support scientists and their organisations. In Chapter 7 we reflect on our choices of implementation technology in light of our experiences in implementing the workflows described here. SubSift implementations of these workflows constitute a proof of concept realisation of this general profiling and matching framework. To the best of our knowledge, despite the potential utility of such a set of services and notwithstanding that the underlying techniques are well established, such an engineering solution is not immediately available elsewhere. Furthermore, one of the demonstrator applications is currently used as a recommender system by editors of two leading computer science journals.

Chapter 4

Higher-Order Dataflows

In this chapter we explore a further abstraction of submission sifting that leads us to propose a higher-order dataflow model that is intended to be more flexible and generalised than the SubSift model. The model ranges over a class of terms in a strongly typed higher-order logic that have previously been shown to be sufficiently expressive to represent a wide variety of unstructured, semi-structured and structured data [Llo03, GLF04]. We show that, under certain assumptions, our higher-order dataflow model can be proven to be both highly parallelisable and potentially scalable to large datasets. These characteristics, taken together with the representational power of the knowledge representation, leads us to suggest that the model could in future be applied to Big Data problems, particularly those involving heterogeneous data, so-called Big Variety data. To investigate the feasibility and utility of this new model we describe JSONMatch, our proof of concept implementation produced by generalising the SubSift framework. Using JSONMatch we demonstrate that the model can in practice be used to implement submission sifting in a more flexible way than the original SubSift implementation. We compare the JSONMatch and SubSift frameworks side-by-side, discussing their relative strengths and weaknesses in different settings. Finally we assess whether JSONMatch does indeed offer users (developers) a more flexible and generalised framework than SubSift, and in so doing examine the implications of the model's inherent extra levels of abstraction for its users.

Further Abstraction of Generalised Submission Sifting

Our work with SubSift and ExaMiner led us to believe that submission sifting could be further generalised to support a wider range of potential applications. Here we set out the further abstractions of generalised submission sifting that gave us the insight for a more flexible and generalised dataflow model. Once again we follow Prolog syntax, but taking slight liberties with the semantics by endowing Prolog with natural support for sets and higher-order terms containing functions – noting that such features can be emulated in Prolog using, for example, lists for sets and built-in higher-order predicates to evaluate functions embedded inside terms.

Our explanation is set out below as a short sequence of abstractions in the definition of the `subsift/4` predicate that maintain its capacity to express the generalised submission sifting workflow, but without having to rely on "hard-coded" profile-

match predicates. Recall that in Definition 3.1.4, we defined *Generalised Submission Sifting* as the following `subsift/4` predicate.

```
subsift(+Reviewers, +Papers, +C, -Matches) :-
    document(Reviewers, C, D1),
    document(Papers, C, D2),
    profile(D1, C, P1),
    profile(D2, C, P2),
    match(P1, P2, C, Matches).
```

As a first step, we remove the distinction between reviewers and papers (as some of our use cases have already done in practice), recognising that these are both sets of documents. At the same time, we will factor out the `document/2` predicates so that `subsift/4` is defined as follows.

```
subsift(+Documents1, +Documents2, +C, -Matches) :-
    profile(Documents1, C, P1),
    profile(Documents2, C, P2),
    match(P1, P2, C, Matches).
```

In the next step we recognise that profiling involves two passes over the data: a first pass to calculate the number of occurrences of each term and to compute the overall totals; and a second pass to compute the tf-idf using the totals in normalisation.

```
subsift(+Documents1, +Documents2, C, -Matches) :-
    preprofile(Documents1, C, PP1), profile(PP1, C, P1),
    preprofile(Documents2, C, PP2), profile(PP2, C, P2),
    match(P1, P2, C, Matches).
```

In the final step, we assume that the counting of terms and computation of the totals in `preprofile/3` and the computation `profile/3` are specified in the context parameter `C`. This relies on `C` being a high-order parameter containing functions capable of expressing these computations. Having refactored the calculation of profile information into `C`, both the `preprofile/3` and `profile/3` predicates can be abstracted to be map functions – that apply functions specified within `C` to each input element to produce each output element. The same technique can be used to refactor `match/4` so that it becomes a Cartesian product function. This all necessitates that `C` contain higher-order components for each of the map and product predicates, resulting in the following definition.

Definition 4.0.1 (Abstracted Submission Sifting) Abstracted Submission Sifting *is a predicate* `subsift/4` *with the definition,*

```
subsift(+Documents1, +Documents2, [+CPP,+CP,+CM], -Matches) :-
    map(Documents1, CPP, PP1), map(PP1, CP, P1),
    map(Documents2, CPP, PP2), map(PP2, CP, P2),
    product(P1, P2, CM, Matches).
```

where input variables `Documents1` *and* `Documents2` *are unified with the sets of representations of documents. Input variables* `CPP`*,* `CP` *and* `CM` *are each unified with some function for pre-profile, profile and match respectively. Output variable* `Match` *is unified with a list of document-document pairs ranked by descending similarity.*

This definition has some interesting characteristics that, as will become clear over the remaining sections of this chapter, overcome the limitations of generalised submission sifting. For the time being we will just remark that the behaviour of the workflow is now substantially defined by the higher-order context parameter: for instance, enabling entirely different profiling and similarity algorithms to be passed in as functions, replacing tf-idf and cosine similarity. In the next section we take this abstraction process to its ultimate conclusion by abstracting the body of the `subsift/4` predicate to be another context parameter.

4.1 A Higher-Order Dataflow Model

In this section we introduce a dataflow model inspired by abstracted submission sifting. The new model allows users to specify more functionality at runtime and potentially opens up new applications involving data types other than text. In preparation for a more formal description, we first give an intuitive overview of the main characteristics of our proposed dataflow model.

The domain and codomain for dataflows in the model are relations. For the purposes of this introduction we define a *relation* to be a set of key-value pairs from $String \times \mathcal{D}$, for some domain $\mathcal{D}$. With suitable choices for $\mathcal{D}$, relations can represent tables from relational databases, documents/values from NoSQL databases, data objects from object databases and graph data from triple stores.

Indeed, we note that implementations of such databases are often underpinned by key-value stores such as the well-known BerkeleyDB, or by filename-file equivalents in distributed file systems where the filename behaves as the key and the file content as the value. From this it is clear that that simple definition of a relation is sufficient to represent a wide variety of unstructured, semi-structured and structured data. Our dataflow model thus ranges over a broad and useful range of data types.

The building blocks of a dataflow are transformations. A *transformation* is a higher-order function of the form $\Phi(g)(h)(s,t,u,\dots)$, ranging over some relations $s,t,u,\dots$, with behaviour determined by functions g and h, the higher-order parameters. A higher-order function has at least one function in either, or both, of its domain and codomain. Parameter g is a function that selects elements from input relations $s,t,u,\dots$ and passes them as inputs to function h, the *data constructor* responsible for creating elements in the transformation's output relation. We deliberately choose to define g and h as two separate parameters to emphasise their individual roles even though they could trivially be combined into a single higher-order parameter. We will revisit parameters g and h to describe them in more detail later in this chapter. Figure 4.1 depicts a prototypical transformation that applies $\Phi(g)(h)$ to the input relations $s,t,u,\dots$ to produce the output relation v.

In preparation for the definition of a dataflow, we define the (reverse) *composition* function, $\circ$, such that $(f_1 \circ f_2)(x) = f_2(f_1(x))$, where $f_1 : \alpha \to \beta$ and $f_2 : \beta \to \gamma$

are functions, and α, β, γ are types, with x having type α. Reverse composition is right associative. Thus $f_1 \circ \cdots \circ f_n = f_1 \circ (f_2 \circ \cdots \circ (f_{n-1} \circ f_n) \cdots)$. For brevity we will refer to reverse composition as simply *composition*, following the convention of the higher-order logic[1] introduced later.

A *higher-order dataflow* is a composition of one or more transformations. A dataflow $\Phi_1 \circ \cdots \circ \Phi_k$ composed of k transformations is referred to as having *size* k. Each Φ_i, $i = 1, \ldots, k$, in a size k dataflow has an associated pair of functions, g_i and h_i that determine the transformation from the input relations to the output relation. The single transformation in Figure 4.1 also constitutes a dataflow of size 1. Compositions of Φ with $k > 1$ enable the representation of arbitrarily complex (acyclic) dataflows with arbitrarily many input relations and a single resulting output relation. All their transformations, apart from the last one, each produce an intermediate relation. The *order* of a dataflow is its total number of relations, which is the sum of the input, output and intermediate relations.

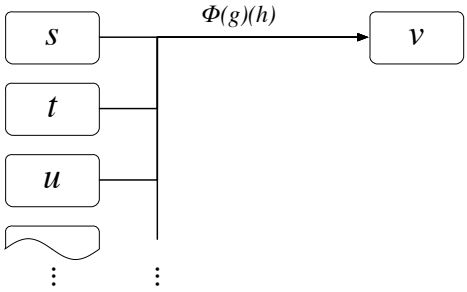


Figure 4.1: Transformation $\Phi(g)(h)(s,t,u,\ldots) = v$.

Figure 4.2 depicts an example dataflow $\Phi_3(\Phi_1(s), \Phi_2(t)) = w$, with intermediate relations u, v, size 3 and order 5. The meaning of Φ_1 is defined by its higher-order parameters, $(h_1)(g_1)$, shown as *filter* in Figure 4.2 for readability. Similarly, Φ_2's parameters $(h_2)(g_2)$ mean *join* and Φ_3's $(h_3)(g_3)$ mean *group*. The meaning of the overall dataflow is to group the result of a join of t with a filtered subset of s.

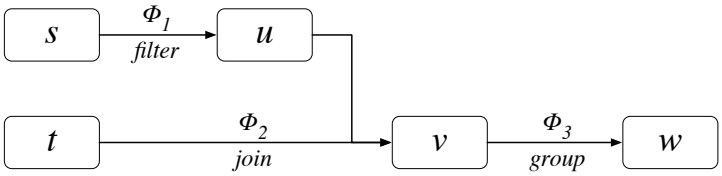


Figure 4.2: Dataflow $\Phi_3(\Phi_1(s), \Phi_2(t)) = w$.

Figure 4.3 depicts an example of a MapReduce [DG08] dataflow $\Phi_5(\Phi_4(\Phi_1(s)), \Phi_4(\Phi_2(s)), \Phi_4(\Phi_2(s))) = z$, with intermediate relations t, u, v, w, x, y, size 7 and order 8. The meaning of $(h_1)(g_1), \ldots, (h_3)(g_3)$ is *split* the data into one of

[1] To enable compositions on n-ary functions for $n > 1$, e.g. $f_3(f_2(x), f_1(x))$, we assume the availability in the logic of a restriction function $\restriction$, where $f \restriction_A : A \to \beta$ given that $f : \alpha \to \beta$ and $A \subseteq \alpha$. For example, the function returned when some argument x_i of function f is substituted by function g is $f \restriction_{x_i = g} = f(x_1, \ldots, x_{i-1}, g(x_1, x_2, \ldots, x_n), x_{i+1}, \ldots, x_n)$.

three partitions, $(h_4)(g_4)$ is *map* and $(h_5)(g_5)$ is *reduce*. The meaning of the overall dataflow is a MapReduce from s to z. In contrast to Figure 4.2, where the dataflow is a tree with data flowing from the leaves to the root, the MapReduce example is a directed acyclic graph (DAG) where the data is split and later joined back together. However, semantically, Figure 4.3 can be redrawn as a tree with relation s occurring three times: once each as the input to t, u and v. This underlying tree structure to the DAG is evident from the multiple appearances of s in the expression $\Phi_5(\Phi_4(\Phi_1(s)),\Phi_4(\Phi_2(s)),\Phi_4(\Phi_3(s)))$.

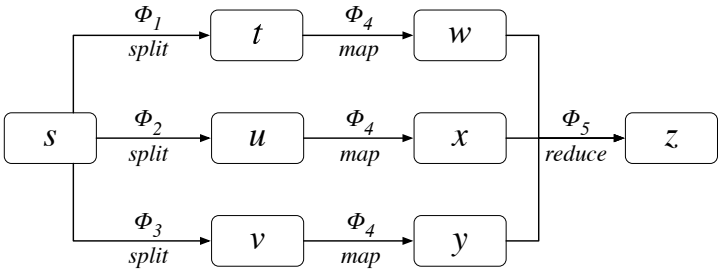


Figure 4.3: Dataflow $\Phi_5(\Phi_4(\Phi_1(s)),\Phi_4(\Phi_2(s)),\Phi_4(\Phi_3(s))) = z$.

Having introduced the high-level structure of a dataflow and its component transformations, we now give a similar overview of generator function g and data constructor h, the higher-order parameters of Φ.

4.1.1 Generator Function (g)

In a transformation of the form $\Phi(g)(h)(s,t,u,\dots)$, *generator function* g ranges over the transformation's input relations. The role of g is to enumerate the elements of $s \times t \times u \times \dots$, assembling them as the inputs to data constructor h, which constructs the elements of the transformation's output relation. Generator g is chosen from $\{\rightarrow, \times, \lambda\}$, referred to as *map*, *product* and *lambda* respectively.

Map ($\rightarrow$) Figure 4.4 depicts examples of three higher-order *map* transformations $\Phi(g)(h)$, where generator function $g = \rightarrow$. In the first, $\Phi(\rightarrow)(h)(s) = v$, the transformation applies h to the elements of s to construct the $n = |s|$ elements of v. In the second, $\Phi(\rightarrow)(h)(s,t) = v$, the transformation applies h to pairs (s_i,t_i), $i = 1\dots n$, to construct the $n = |s| = |t|$ elements of v. There is a bijection between each source relation and the target relation v. In the third, a composition of multiple separate *map* transformations is an m:1 injective mapping from the elements of relations $s,t,u,\dots$ to the $|s|+|t|+|u|+\dots$ elements of relation v. The intuition of this composition is that v is the union of relations $s,t,u,\dots$, which is a useful idiom of the model.

Product ($\times$) Figure 4.5 depicts an example higher-order *product* transformation $\Phi(g)(h)$, where generator function $g = \times$. The transformation $\Phi(\times)(h)(s,t) = v$ applies h to each element in $s \times t$ to construct the $|s \times t| = |s| \cdot |t|$ elements of v.

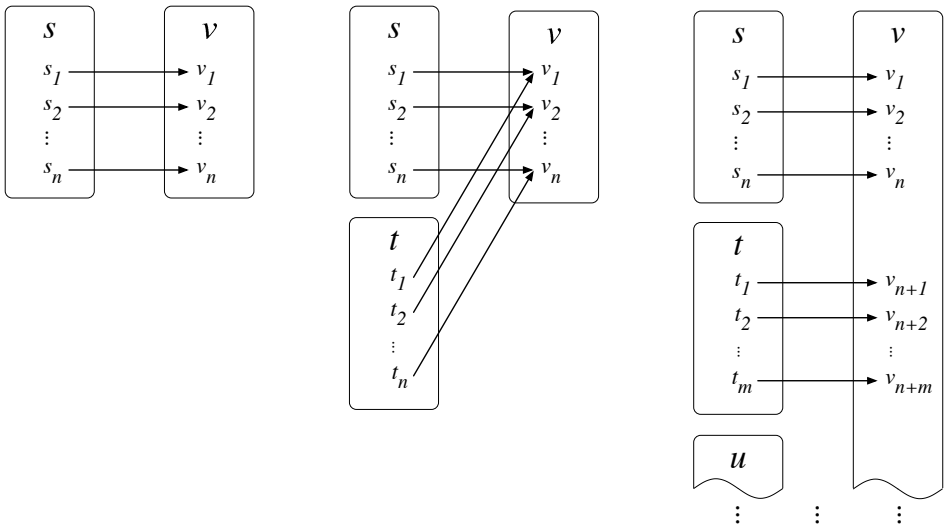


Figure 4.4: Three kinds of higher-order *map* transformations: left, $\Phi(\rightarrow)(h)(s) = v$; centre, $\Phi(\rightarrow)(h)(s,t) = v$; and right, a transformation formed by the composition of multiple *map* transformations, $\Phi(\rightarrow)(h)(s) \circ \Phi(\rightarrow)(h)(t) \circ \Phi(\rightarrow)(h)(u) \circ \cdots = v$.

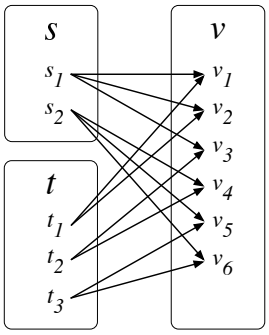


Figure 4.5: Higher-order *product* transformation $\Phi(\times)(h)(s,t) = v$.

Lambda (λ) Figure 4.6 depicts an example higher-order *lambda* transformation $\Phi(g)(h)$, where generator function $g = \lambda$. Unlike $\rightarrow$ and $\times$, the λ generator has a parameter which controls its behaviour. The parameter ℓ is application-specific and there is no default in the model, so it must be supplied. Intuitively, ℓ is a function that splits an element of a relation into multiple sub-parts in some useful way for the particular application at hand, e.g. decomposing a set into its elements or decomposing a string into substrings. The lambda transformation $\Phi(\lambda)(h)(s) = v$ generates multiple sub-parts, x_j, of s_i using function ℓ and then applies $\lambda x_j . h$ to s_i for each x_j such that $v = \{v_j : v_j = \lambda x_j . h(s_i),\ x_j \in \ell(s_i),\ s_i \in s\}$. The example in Figure 4.6 assumes a choice of function ℓ that generates multiple elements $v_j \in v$ from each $s_i \in s$. However, different choices of ℓ could result in a single or even zero v_j from each s_i. The number of v_j generated from each s_i may vary and so, in the example, it is not necessary that $m = o = \ldots$, nor that $m, o, \cdots > 0$. Also, when $\{v_1, \ldots, v_m\} \cap \{v_{m+1}, \ldots, v_{m+o}\} \cap \cdots \neq \varnothing$, more than one s_i may generate (or

contribute to the value of) the same s_i, enabling a M:M mapping between s and v.

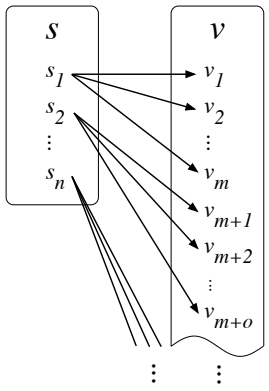


Figure 4.6: Higher-order *lambda* transformation $\Phi(\lambda)(h)(s) = v$.

4.1.2 Data Constructor (h)

In a transformation $\Phi(g)(h)$, *data constructor* function h ranges over the outputs of generator function g. Thus in the case of $g \in \{\rightarrow, \times\}$, the inputs to h will be elements from the input relations to Φ. In the case of $g = \lambda$, the inputs to h can be sub-parts of the elements from the input relations to Φ. The role of h is to construct the elements of the output relation of the transformation. The result of each invocation of h is an element in the output relation.

4.2 Knowledge Representation

Our dataflow model follows the *individuals-as-terms* knowledge representation, a generalisation of the relational model's attribute-value representation, that collects all information about an individual in a single term (or document). The terms are the *basic terms* from a family of typed terms in the higher-order logic based on Church's simple theory of types with several extensions [Llo02]. The logic natively supports data types that are important for representing individuals, including sets, multisets and graphs. Strong typing helps to reduce search spaces and the type of terms provides useful metadata. Also, type polymorphism and type hierarchies are powerful tools in the representation and processing of knowledge. The theory behind the logic and the individuals-as-terms formalism is set out in [Llo02] and is summarised in Chapter 2, which includes examples of basic term representations of types such as sets, multisets, lists and trees. The following two examples demonstrate the expressive power of the formalism, showing that relations (tables) from Codd's relational model and relations (outer bags) as defined in Apache Pig may also be elegantly and concisely expressed using this formalism [PF08, PF13c].

Example 4.2.1 (Relational Tables) *In the original relational model, a relation R of degree n is a finite set of n-tuples such that $R \subseteq D_1 \times \cdots \times D_n$ where $D_1, \ldots, D_n$*

are domains. By identifying $D_1, \ldots, D_n$ *with types* $\alpha_1, \ldots, \alpha_n \in \mathfrak{B}$ *and defining a type* $\mathfrak{B}_{Table}$*, basic terms can be used to represent relational tables as*

$$\mathfrak{B}_{Table} = \mathfrak{B}_{\alpha \to \Omega}, \text{ where } \alpha = \mathfrak{B}_{\alpha_1 \times \cdots \times \alpha_n}.$$

This simply states that a table is a finite set of basic tuples. We note that practical SQL implementations of relational databases use, so called, unordered labelled tuples where each 'tuple' is a mapping from schema attribute names to attribute values. Duplicate tuples are allowed. This can be represented using basic terms by defining a relational table to be a finite multiset of basic abstractions which map attribute names to attribute values.

Example 4.2.2 (Pig Latin Relations) *In Pig Latin, relations are the main data type and most statements in the language operate on relations. A Pig Latin relation is a bag of tuples. The tuples in a bag are unordered. A bag can have duplicate tuples. A bag can have tuples with differing numbers of fields. Pig permits accessing a tuple field that does not exist and will return a null value. Tuple fields in the same position are not required to be of the same type. By defining the type* $\mathfrak{B}_{Pig}$*, and a data constructor* $\varepsilon \in \mathfrak{B}$ *to represent null, basic terms can be used to represent Pig relations with the required properties.*

$$\mathfrak{B}_{Pig} = \mathfrak{B}_{\alpha \to Nat}, \text{ where } \alpha = \mathfrak{B}_{Nat \to \mathfrak{B}}, \text{ where } s_0 = \varepsilon.$$

In other words, a Pig relation is a multiset of basic abstractions from field number to field value with the default value of null. Field values can be any type of basic term.

In the next section, following on from the above examples, we begin to define our dataflow model by first defining our own representation of relations.

4.3 Formalising the Higher-Order Dataflow Model

We now formally define dataflows in the higher-order logic, beginning with definitions of basic relations and transformations on these relations.

Definition 4.3.1 (Basic Relation) *A* basic relation $r \in \mathfrak{R}_{\beta \to \gamma}$ *is an element in* $\mathfrak{B} \times \mathfrak{B}_{\beta \to \gamma}$*, for some type* $\beta, \gamma \in \mathfrak{B}$*. The universe of* basic relations *is denoted* $\mathfrak{R}$*.*

This states that each basic relation r is a pair (p, c) consisting of a single basic term p and a basic abstraction c from basic terms of type β to basic terms of type γ. The intended meaning is that each r is an object with arbitrary properties represented by p, and a collection of uniformly typed items (our 'individuals') represented by c as shown diagramatically in Figure 4.7. Thus a basic relation uses a basic abstraction c as a generalisation of the set relations from our introduction. A basic term, p, to represent properties of the relation has also been added.

Note that no constraints are placed on the basic term $p \in \mathfrak{B}$ and so any type of basic term may be chosen, for example a key-value property list (i.e. dictionary or hash table). One common use of such a property list in profile matching applications is to hold counts and summary statistics pertaining to the items in c, a technique

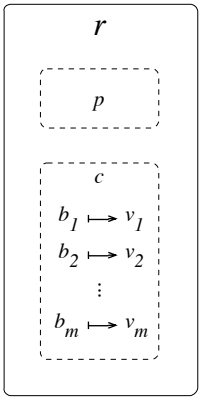


Figure 4.7: Basic relation $r = (p, c)$ where $r \in \mathfrak{R}_{\beta \to \gamma}$ such that $p \in \mathfrak{B}$ and $c \in \mathfrak{B}_{\beta \to \gamma}$ with keys $b_1, b_2, \ldots, b_m \in \beta$ and values $v_1, v_2, \ldots, v_m \in \gamma$, $m \in \mathbb{N}$.

demonstrated in Section 4.4.3. If, on the other hand, p is not required in a specific application then it may be defined to be the empty term ε. The way that p is used in applications has important implications for the parallisability and scalability of the model, a topic we discuss later in Section 4.3.3.

Depending on the choice of β and γ, the collection c in a basic relation of type $\mathfrak{R}_{\beta \to \gamma}$ can represent a wide variety of item types. For example,

- if $\gamma = \Omega$ with default term $s_0 = \bot$, then c is a set;
- if $\gamma = Nat$ and $s_0 = 0$, then c is a multiset (or bag);
- if $\beta = Nat$, then c is a (possibly sparse) array;
- if $\beta = String$, then c is a dictionary (or hash table).

In preparation for our definition of a transformation, let ϕ be a function with signature $\phi : \mathfrak{S}^c \times \mathfrak{B} \times \mathfrak{B} \to \mathfrak{B}$ and definition $\phi(\tau, u, \rho) = v$, denoted $\phi_\tau(u, \rho)$, where τ is a term containing zero or more functions on (u, ρ) and result v is τ after substituting any contained functions with the result of their application on (u, ρ). Intuitively, ϕ expands functions embedded in 'template' term τ such that each of those functions accepts (u, ρ) as inputs and produces a single basic term as its output.

We recall from the definition of *support* in Chapter 2 (Definition 2.3.19, p.58) that for a given basic abstraction $t \in \mathfrak{B}_{\beta \to \gamma}$, $\text{supp}(t)$ is the set of keys in t. Thus, for example, in Figure 4.7 the support of c is $\text{supp}(c) = \{b_1, b_2, \ldots, b_m\}$. Similarly, for a multiset $t \in \mathfrak{B}_{Q \to \text{Nat}}$ where $t = \langle (A, 3), (B, 1), (C, 2) \rangle$ and $A, B, C : Q$, for some type $Q \in \mathfrak{B}$, then $\text{supp}(t) = \{A, B, C\}$.

Recall also that the definition of support employs a function V that is used to index items in the basic abstraction such that the application $V(t\ b)$ returns the value for some key $b \in \beta$, i.e. in our example $V(t\ A) = 3$, $V(t\ B) = 1$, $V(t\ C) = 2$ and $V(t\ b) = 0$ for all $b \notin \{A, B, C\}$. An alternative syntax for the same concept might have been to dispense with V as an explicit function and instead write t_b (e.g. $t_\text{A} = 3$) but doing so for the following definition would have resulted in two levels of subscript. So, we retain the V notation in our definition of transformation

but emphasise that where $V(c\ k)$ appears on the extreme left of the large multicase equation it is simply being used to address element c_k in the basic abstraction c.

Definition 4.3.2 (Transformation) *For some basic relations,* $\mathfrak{R}_1,\ldots,\mathfrak{R}_n \in \mathfrak{R}$, *a* transformation Φ *is a higher-order function with signature,*

$$\Phi : \mathfrak{B}_G \to \mathfrak{B}_H \to \mathfrak{R}_1 \to \cdots \to \mathfrak{R}_n \to \mathfrak{R},$$

and definition, $\Phi(g)(h)(t_1,\ldots,t_n) = (p,c)$, *where parameters* $g \in \{\to,\times,\lambda\}$ *and* $h = (\tau_{Relation},\tau_{Item},\tau_{Lambda})$, *for some template terms* $\tau_{Relation},\tau_{Item},\tau_{Lambda} \in \mathfrak{S}^c$, *and* $t_1,\ldots,t_n = (p_1,c_1),\ldots,(p_n,c_n)$. $p = \phi_{Relation}(t_1,\ldots,t_n)$. *For* c, *there are the following three cases, depending on* g.

$$V(c\ k) = \begin{cases} \phi_{Item}((V(c_1\ k),\ldots,V(c_n\ k)),(p_1,\ldots,p_n)), & \\ \quad \forall k \in supp(c_1) & \text{if } g = \to, \\ \phi_{Item}((V(c_1\ b_1),\ldots,V(c_n\ b_n)),(p_1,\ldots,p_n)), & \\ \quad k = (b_1,\ldots,b_n), & \\ \quad \forall b_1 \in supp(c_1),\ldots,\forall b_n \in supp(c_n) & \text{if } g = \times, \\ \lambda x.\,\phi_{Item}((V(c_1\ b),\ldots,V(c_n\ b)),(p_1,\ldots,p_n)), & \\ \quad \forall (k,x) \in \phi_{Lambda}((V(c_1\ b),\ldots,V(c_n\ b)),(p_1,\ldots,p_n)), & \\ \quad \forall b \in supp(c_1) & \text{if } g = \lambda. \end{cases}$$

The three cases for c above are broadly similar to their respective cases from our introduction, but are generalised to use any type of basic abstractions rather than just sets. If $g \in \{\to,\lambda\}$, then abstractions $c_2,\ldots,c_n$ must include the same keys as c_1, such that $\text{supp}(c_1) \subseteq \text{supp}(c_2) \cap \cdots \cap \text{supp}(c_n)$. No such constraint on the keys exists for $g = \times$, by the definition of the Cartesian product.

In every application of $\phi_{Relation}$ and ϕ_{Item}, any embedded functions in $\tau_{Relation}$ and τ_{Item} respectively are expanded (i.e. each embedded function is evaluated and then substituted in the template with its result). The same also applies in the case when $g = \lambda$, where any embedded functions in τ_{Lambda} are expanded prior to every evaluation of ϕ_{Lambda}. The result of each application of ϕ_{Lambda} generates a (k,x) pair, the x from which is then lambda substituted into any occurrences of x within embedded functions in τ_{Item}, resulting in the application of $\lambda x.\,\phi_{Item}$ to the arguments, rather than just ϕ_{Item} as would be the case if $g \in \{\to,\times\}$. It is worth noting with $\lambda x.\,\phi_{Item}$ that expressions embedded within τ_{Item} do not have to refer to the arguments $(V(c_1\ b),\ldots,V(c_n\ b)),(p_1,\ldots,p_n)$ if those arguments are not required, for example in cases where the outputs $V(c\ k)$ are computed entirely by ϕ_{Lambda}.

For the time being we will leave open the nature of the functions embedded within the template terms in $h \in \mathfrak{B}_H$ as, for the most part, the choice of these functions is a matter for a specific implementation and not a part of our model per se. The one exception to this is a class of functions called *path expressions* which are required by our model and are formally defined in next section. However, concrete examples of an application-specific range of embedded functions and their use in template terms

are given in "Section 4.4 – Implementation".

To help explain Definition 4.3.2 we give schematics of the computation of a transformation for each of the three cases: $g = \rightarrow$, Figure 4.8; $g = \times$, Figure 4.9; and $g = \lambda$, Figure 4.10. We also give examples of input and output relations for three such transformations, as Examples 4.3.3, 4.3.4 and 4.3.5 respectively.

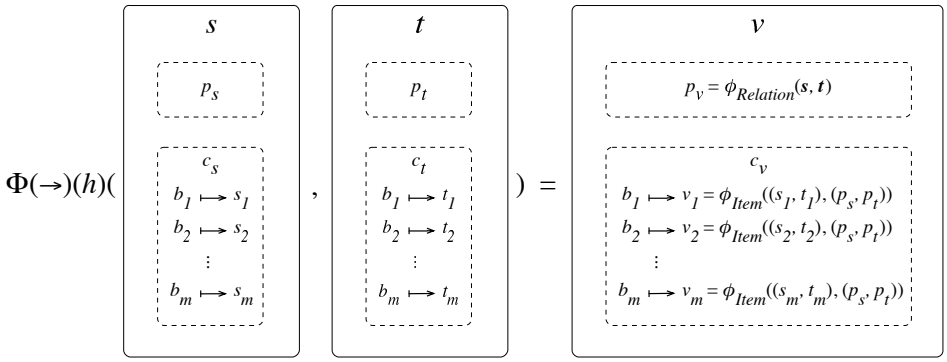


Figure 4.8: Map transformation $\Phi(\rightarrow)(h)(s,t) = v$ where $s,t,v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \varepsilon)$ and ε is the empty term.

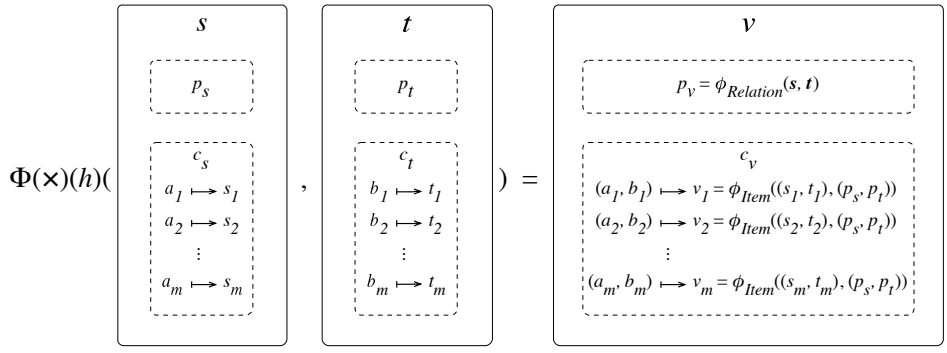


Figure 4.9: Product transformation $\Phi(\times)(h)(s,t) = v$ where $s,t,v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \varepsilon)$ and ε is the empty term.

Example 4.3.3 (Map transformation) *Let $s,t,v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \varepsilon\}$ and e_1, e_2, e_3 be expressions containing embedded functions. Let e_1 be an expression that counts the total number of words in all the strings $s_i \in c_s$ and $t_i \in c_t$. Let e_2 be an expression that concatenates a string from s_i with a string from t_i. Let e_3 be an expression that counts the combined total number of words in s_i and t_i. For the values of s,t and h shown below, the map transformation $\Phi(\rightarrow)(h)(s,t)$ results in*

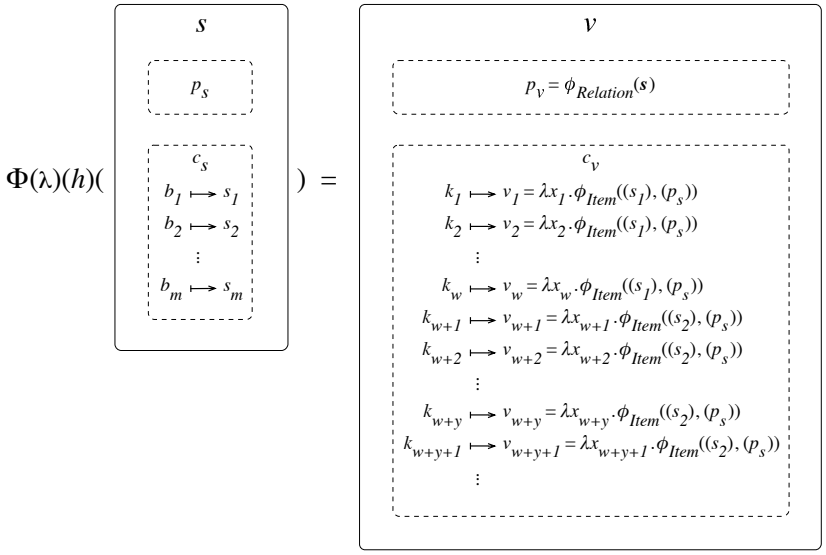


Figure 4.10: Lambda transformation $\Phi(\lambda)(h)(s) = v$ where $s, v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \tau_{Lambda})$. For each s_i, where $i \in \{1, \ldots, m\}$, generator function ϕ_{Lambda} results in $j_i \in \mathbb{N}$ pairs (k, x) such that $k \in \mathfrak{B}$ is some key and $x \in \mathfrak{S}^c$ is some sub-part of s_i.

v as follows.

$$
\begin{aligned}
s &= (\varepsilon, \{b_1 \mapsto \text{“}quick\ brown\text{”}, b_2 \mapsto \text{“}lazy\text{”}\}), \\
t &= (\varepsilon, \{b_1 \mapsto \text{“}fox\text{”}, b_2 \mapsto \text{“}dog\text{”}\}), \\
\tau_{Relation} &= \{words \mapsto e_1\}, \\
\tau_{Item} &= [e_2, e_3], \\
v &= (\{words \mapsto 5\}, \{b_1 \mapsto [\text{“}quick\ brown\ fox\text{”}, 3], b_2 \mapsto [\text{“}lazy\ dog\text{”}, 2]\}).
\end{aligned}
$$

Example 4.3.4 (Product transformation) *Let $s, t, v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \varepsilon\}$ and e_1, e_2 be expressions containing embedded functions. Let e_1 be an expression that counts the number of matching pairs of strings $s_i \in c_s$ and $t_i \in c_t$. Let e_2 be an expression that compares a string from s_i with a string from t_i, returning $\top$ (true) if they match or $\bot$ (false) otherwise. For the values of s, t and h shown below, the map transformation $\Phi(\times)(h)(s,t)$ results in v as follows.*

$$
\begin{aligned}
s &= (\varepsilon, \{a_1 \mapsto \text{“}Wallace\text{”}, a_2 \mapsto \text{“}Gromit\text{”}, a_3 \mapsto \text{“}Shaun\text{”}\}), \\
t &= (\varepsilon, \{b_1 \mapsto \text{“}Gromit\text{”}, b_2 \mapsto \text{“}Wallace\text{”}\}), \\
\tau_{Relation} &= \{matches \mapsto e_1\}, \\
\tau_{Item} &= e_2, \\
v &= (\{matches \mapsto 2\}, \{(a_1, b_1) \mapsto \bot, (a_1, b_2) \mapsto \top, (a_2, b_1) \mapsto \top, (a_2, b_2) \mapsto \bot, \\
&\qquad (a_3, b_1) \mapsto \bot, (a_3, b_2) \mapsto \bot\}).
\end{aligned}
$$

Example 4.3.5 (Lambda transformation) *Let $s, v \in \mathfrak{R}$, $h = (\tau_{Relation}, \tau_{Item}, \tau_{Lambda}\}$ and e_1, e_2, e_3 be expressions containing embedded functions. Let e_1 be an expres-*

sion that returns a sorted list of words from all strings $s_i \in c_s$. Let e_2 be an expression that returns a unique identifier v_j, $j \in \mathbb{N}$. Let e_3 be an expression that splits a string from $s_i \in c_s$ into words. For the values of s and h shown below, the map transformation $\Phi(\lambda)(h)(s)$ results in v as follows.

$$
\begin{aligned}
s &= (\varepsilon, \{b_1 \mapsto \text{“}\textit{quick brown fox}\text{”}, b_2 \mapsto \text{“}\textit{lazy dog}\text{”}\}),\\
\tau_{Relation} &= e_1,\\
\tau_{Item} &= x,\\
\tau_{Lambda} &= (k = e_2, x = e_3),\\
v &= ([\text{“}\textit{brown}\text{”}, \text{“}\textit{dog}\text{”}, \text{“}\textit{fox}\text{”}, \text{“}\textit{quick}\text{”}, \text{“}\textit{lazy}\text{”}],\\
&\quad \{v_1 \mapsto \text{“}\textit{quick}\text{”}, v_2 \mapsto \text{“}\textit{brown}\text{”}, v_3 \mapsto \text{“}\textit{fox}\text{”}, v_4 \mapsto \text{“}\textit{lazy}\text{”}, v_5 \mapsto \text{“}\textit{dog}\text{”}\}).
\end{aligned}
$$

We define the higher-order dataflows of our model by generalising the definition from our introduction.

Definition 4.3.6 (Higher-Order Dataflow) *A* higher-order dataflow *is a composition of transformations on basic relations.*

To complete our formal definition of the model we now turn to the required class of embedded functions called *path expressions*. As preparation for the definition of a path expression we first introduce the concept of a *path*.

4.3.1 Paths

When defining the higher-order term h in transformations, $\Phi(g)(h)$, it is useful to be able to access the value of a subterm at one or more specific positions within a term, for example, an element in a set or the n-th item of every tuple in a set of tuples. The logic includes a scheme for enumerating subterms and referring to a subterm at a specific position by a string label [Llo03] but that scheme is not suitable in our context because labelling varies from term to term. A type-based enumeration of terms addresses this problem but at the cost of not uniquely labelling every subterm [PF08]. Here we adopt a path-based approach that uniquely labels every subterm but without the restrictions of type-based labelling.

Informally, a *path* is a unique label for a subterm. The set of all such unique labels for a term is called the *path set* of the term. More formally, we can define the concept of a path by first defining the path set of a term as follows. Let $\mathbb{Z}^+$ denote the set of positive integers and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string. Let $\mathfrak{L}_R$ denote the set of reserved characters, $\{\$, .,], [, \star\}$. Let $\mathfrak{L}_L = \mathfrak{L} - \mathfrak{L}_R - (\mathbb{Z}^+)^*$ denote the set of labels. Let $\mathfrak{L}_P = (\mathbb{Z}^+)^* \cup (\mathfrak{L}_R) \cup (\mathfrak{L}_L)^*$, and $(\mathfrak{L}_P)^*$ denote the set of all strings over the alphabet $\mathfrak{L}_P$. Assuming the existence of the usual concatenation function, $1p$ denotes the string concatenation of 1 with p, ℓp denotes the concatenation of ℓ with p, where label $\ell \in (\mathfrak{L}_L)^*$. Let uniq_β be a function, with signature $\text{uniq}_\beta : \beta \to \textit{String}$, that maps $t \in \beta$ to a unique string ℓ_t such that $\ell_t = \ell_s \implies t = s$, for all $t, s \in \beta$.

Definition 4.3.7 (Path set) *The* path set *of a term* $t \in \mathfrak{S}^c$, *denoted* $\mathcal{P}(t)$, *is a finite set of symbols drawn from the set of strings* $(\mathfrak{L}_P)^*$, *defined inductively as follows.*

1. *If* t *is a variable, then* $\mathcal{P}(t) = \{\varepsilon\}$.
2. *If* t *is a constant, then* $\mathcal{P}(t) = \{\varepsilon\}$.
3. *If* t *has the form,*

   $$\lambda x.\, \textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0$$

   then $\mathcal{P}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{\,.\,\ell_i p_i \mid \ell_i = uniq_\beta(t_i),\ p_i \in \mathcal{P}(s_i)\}$.
4. *If* t *has the form* $(u\ v)$,
   then $\mathcal{P}(t) = \{\varepsilon\} \cup \{1p \mid p \in \mathcal{P}(u)\} \cup \{2p' \mid p' \in \mathcal{P}(v)\}$.
5. *If* t *has the form* $(t_1, \ldots, t_n)$,
   then $\mathcal{P}(t) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{\,[i]\,p_i \mid p_i \in \mathcal{P}(t_i)\}$.

Each $p \in \mathcal{P}(t)$ *is called a* path *of* t.

Definition 4.3.8 (Subterm at path) *If* $t \in \mathfrak{S}^c$ *is a term and* $p \in \mathcal{P}(t)$ *then, the* subterm of t at path p, *denoted* $t|_p$, *is defined inductively on the length of* p *as follows.*

1. *If* $p = \varepsilon$, *then* $t|_p = t$.
2. *If* $p = .\ell_i p'$, *for some* p', *and* t *has the form,*

   $$\lambda x.\, \textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0$$

   then $t|_p = (t_i, s_i)|_{p'}$, *where* $\ell_i = uniq_\beta(t_i)$ *and* $t_i \in \beta$, *for* $i = 1, \ldots, n$.
3. *If* $p = 1p'$, *for some* p', *and* t *has the form* $(u\ v)$, *then* $t|_p = u|_{p'}$.
4. *If* $p = 2p'$, *for some* p', *and* t *has the form* $(u\ v)$, *then* $t|_p = v|_{p'}$.
5. *If* $p = [i]\,p'$, *for some* p', *and* t *has the form* $(t_1, \ldots, t_n)$,
   then $t|_p = t_i|_{p'}$, *for* $i = 1, \ldots, n$.

A subterm *is* proper *if it is not at path* ε.

Notice from Part 2 of Definition 4.3.8 that the path associated with an abstraction identifies the key-value pair rather than just the value. The key-value pair is itself a tuple and therefore its items are individually associated with paths by Part 5 of the definition. Table 4.1 gives some examples of terms, their path sets and associated subterms.

Proposition 4.3.9 *Each subterm at a basic path is a term.*

Table 4.1: EXAMPLE PATHS

Term t	**Path Set** $\mathcal{P}(t)$	**Subterms** $t\|_{p\in\mathcal{P}(t)}$
(A,B,C)	$\{\varepsilon,$ `[1]`, `[2]`, `[3]`$\}$	t,A,B,C
$\{P,Q\}$	$\{\varepsilon,$`p`,`p[1]`,`p[2]`,`q`,`q[1]`,`q[2]`$\}$	$t,(P,\top),P,\top,(Q,\top),Q,\top$
$<A,A,A,B,B>$	$\{\varepsilon,$`a`,`a[1]`,`a[2]`,`b`,`b[1]`,`b[2]`$\}$	$t,(A,3),A,3,(B,2),B,2$
$[A,B,C]^{\dagger}$	$\{\varepsilon,$`1`,`2`,`21`,`22`,`221`,`222`$\}$	$t,A,[B,C],B,[C],C,[]$
$(A,\{P\},B,C)$	$\{\varepsilon,$`[1]`, `[2]`, `[2].p`, `[2].p[1]`, `[2].p[2]`, `[3]`, `[4]`$\}$	$t,A,\{P\},(P,\top),P,\top,B,C$

† Assuming t is represented as depicted in Figure 2.5 (p.54).

Proof 4.3.10 *Let t be a term and $p \in \mathcal{P}(t)$. It is shown by induction on the length of p that $t|_p$ is a term.*

If the length of p is 0, then $p = \varepsilon$. Thus $t|_p = t$, which is a term. For the inductive step, suppose the length of p is $k+1$ $(k \geqslant 0)$. There are several cases to consider.

If $p = .\ell_i p'$, for some p', and t has the form,

$$\lambda x.\, \mathit{if}\ x = t_1\ \mathit{then}\ s_1\ \mathit{else}\ \ldots\ \mathit{if}\ x = t_n\ \mathit{then}\ s_n\ \mathit{else}\ s_0$$

then $t|_p = (t_i, s_i)|_{p'}$, which is a term by the induction hypothesis, where $\ell_i = uniq_\beta(t_i)$ and $t_i \in \beta$, for $i = 1,\ldots,n$.

If $p = 1p'$, for some p', and t has the form $(u\ v)$, then $t|_p = u|_{p'}$, which is a term by the induction hypothesis.

If $p = 2p'$, for some p', and t has the form $(u\ v)$, then $t|_p = v|_{p'}$, which is a term by the induction hypothesis.

If $p = [i]\,p'$, for some p', and t has the form $(t_1,\ldots,t_n)$, then $t|_p = t_i|_{p'}$, which is a term by the induction hypothesis, for $i = 1,\ldots,n$.

Proposition 4.3.11 *Each subterm at a basic path is a basic term.*

Proof 4.3.12 *Trivial from Proposition 4.3.9 by the definition of a basic term.*

4.3.2 Path Expressions

A *path expression* extends the concept of a path by introducing recursive descent ('..') and wildcard ('$\star$') symbols into the paths of Definition 4.3.7.

Definition 4.3.13 (Path expression) *The set of* path expressions *of a term $t \in \mathfrak{S}^c$, denoted $\mathcal{P}_e(t)$, is the finite set of symbols drawn from the set of strings $(\mathfrak{L}_P)^*$, defined inductively as follows.*

1. *If* t *has the form,*

$$\lambda x.\, \textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0$$

then $\mathcal{P}_e(t) = \mathcal{P}(t) \cup \bigcup_{i=1}^{n} \{\varsigma p_i \mid \varsigma \in \{.\,., .*\},\ p_i \in \mathcal{P}_e(s_i)\}$.

2. *If* t *has the form* $(t_1, \ldots, t_n)$,
then $\mathcal{P}_e(t) = \mathcal{P}(t) \cup \bigcup_{i=1}^{n} \{\varsigma p_i \mid \varsigma \in \{.\,., [*]\},\ p_i \in \mathcal{P}_e(t_i)\}$.

3. *Otherwise,* $\mathcal{P}_e(t) = \mathcal{P}(t)$.

Each $q \in \mathcal{P}_e(t)$ *is called a* path expression *on* t.

A path, $p \in \mathcal{P}(t)$, identifies a single subterm of t, whereas a path expression, $q \in \mathcal{P}_e(t)$, matches some subset of paths in $\mathcal{P}$, consequently identifying none or more subterms of t.

Definition 4.3.14 (Subterms at path expression) *If* $t \in \mathfrak{S}^c$ *is a term and* $q \in \mathcal{P}_e(t)$ *then, the* subterms of t at path expression q, *denoted* $t|_q$, *is defined inductively on the length of* q *as follows.*

1. *If* $q = \varepsilon$, *then* $t|_q = t$.

2. *If* $q = .\ell_i q'$, *for some* q', *and* t *has the form,*

$$\lambda x.\, \textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0$$

then $t|_q = (t_i, s_i)|_{q'}$, *where* $\ell_i = \mathit{uniq}_\beta(t_i)$ *and* $t_i \in \beta$, *for* $i = 1, \ldots, n$.

3. *If* $q = 1q'$, *for some* q', *and* t *has the form* $(u\ v)$, *then* $t|_q = u|_{q'}$.

4. *If* $q = 2q'$, *for some* q', *and* t *has the form* $(u\ v)$, *then* $t|_q = v|_{q'}$.

5. *If* $q = [i]q'$, *for some* q', *and* t *has the form* $(t_1, \ldots, t_n)$,
then $t|_q = t_i|_{q'}$, *for* $i = 1, \ldots, n$.

6. *If* $q = .*q'$, *for some* q', *and* t *has the form,*

$$\lambda x.\, \textit{if } x = t_1 \textit{ then } s_1 \textit{ else } \ldots \textit{ if } x = t_n \textit{ then } s_n \textit{ else } s_0$$

then $t|_q = ((t_1, s_1)|_{q'}, \ldots, (t_n, s_n)|_{q'})$, *where* $t_i \in \beta$, *for* $i = 1, \ldots, n$.

7. *If* $q = [*]q'$, *for some* q', *and* t *has the form* $(t_1, \ldots, t_n)$,
then $t|_q = (t_1|_{q'}, \ldots, t_n|_{q'})$, $n > 0$.

8. *If* $q = ..q'$, *for some* q', *then* $t|_q = (t|_u, \ldots)$, *for all* $u \in \mathcal{P}_e(q')$.

The result of a path expression on a term is a tuple. Items 1-5 above define the subterm at a path, a single value, that can be considered a 1-tuple. Items 6-8 above define the subterm at one or more paths, collecting all the n values of the matching paths together into an n-tuple. Item 6 is a generalisation of item 2 that matches any value of ℓ. Item 7 is similarly a generalisation of item 2, for tuples. Item 8 is a further generalisation of items 6-7, collecting all values at every subterm into a tuple.

Table 4.2 gives some examples of terms, path expressions and resultant tuple of subterms returned by the application of each expression. The table is divided into two sections, the first of which are example path expressions from the path set of the term, the second where the expressions include recursive descent and wildcard symbols and are not from the path set.

4.3.3 Parallelisability and Scalability of the Model

In this section we prove that a certain class of transformations, which we refer to as *pure transformations*, satisfy the conditions of an *embarrassingly parallel* function and hence, in principle at least, are highly parallelisable [Fos95, WA99]. We also explain why this result also suggests that the transformation of basic relations that exceed the size of working memory are feasible in the model, irrespective of whether computation is carried out in parallel or serially. Taken together, the model's capacity for parallel and scaleable implementation suggests that it may be applicable to Big Data problems involving large datasets – although practical demonstration of this possibility is left to future work. However, later in this chapter we demonstrate (serial) transformation of larger-than-memory basic relations using JSONMatch, our proof of concept implementation of the model.

An embarrassingly parallel problem is one which can be broken down into multiple smaller tasks, each of which may be solved independently, requiring little or no communication between each of these tasks[2]. Thus, a function that expresses a solution to a highly parallelisable problem is a good candidate for decomposing into multiple smaller functions suitable for parallel execution across multiple processors on the same computer or distributed across multiple computers. More formally, we define the concept of *embarrassingly parallel* as follows.

Definition 4.3.15 (Embarrassingly Parallel) *Function* $f : D \rightarrow E$*, for some types* D, E*, is* embarrassingly parallel *if* f *can be decomposed into functions* $f_i : D \rightarrow E$ *such that* $f(d) = \{ f_i(d_i) : d_i \in d,\ i = 1, \ldots, m \}$*, where* $m = |d|$*,* $d \subseteq D$*.*

The key point of the above definition is that computation of $f(d)$ depends on all the elements $d_1, \ldots, d_m$ in input d whereas the computation of $f_i(d_i)$ depends only on the single input d_i. In the context of Big Data problems this has important practical implications in two areas, which we briefly give the intuition of below.

1. **Parallelisability** – Each function $f_i(d_i)$ may be computed independently on a separate processor and its result combined with those of the others into the single result $e = f(d)$. Where the processors all have access to the same memory (working or store) into which the elements of e are to be written then parallel computation can proceed using this shared memory architecture. However, "big volume" Big Data parallel computation often involves distributing copies of data and algorithm across multiple machines with no shared memory – the implementation of MapReduce algorithms being a famous example of such distributed parallelism [DG08]. In this distributed architecture, parallelisability also depends on the amount of data required for each compuation (which

[2]The use of "embarrassingly" here is intended to convey a good thing, not a bad one. This terminology confuses some people and so the name "pleasingly parallel" is also sometimes used.

Table 4.2: EXAMPLE PATH EXPRESSIONS

Term t	**Path Expression** $q \in \mathcal{P}_e(t)$	**Subterms** $t\|_q$
	where $q \in \mathcal{P}(t)$	
(A,B,C)	`t[1]`	(A)
	`t[2]`	(B)
	`t[3]`	(C)
$\{P,Q\}$	`t.p`	$((P,\top))$
	`t.p[1]`	(P)
	`t.p[2]`	$(\top)$
$<A,A,A,B,C,C>$	`t.a`	$((A,3))$
	`t.b`	$((B,1))$
	`t.c`	$((C,2))$
	`t.a[2]`	(3)
	`t.c[1]`	(C)
$(A,\{P,Q\},C,(D,E))$	`t[4]`	$((D,E))$
	`t[4][2]`	(E)
	`t[2]`	$(\{P,Q\})$
	`t[2].q[1]`	(Q)
$[A,B,C]^\dagger$	`t.1`	(A)
	`t.2`	$([B,C])$
	`t.21`	(B)
	`t.22`	$([C])$
	`t.221`	(C)
	`t.222`	$([])$
	where $q \in \mathcal{P}_e(t) \backslash \mathcal{P}(t)$	
(A,B,C)	`t.*`	(A,B,C)
$\{P,Q\}$	`t.*`	$((P,\top),(Q,\top))$
	`t.*[1]`	(P,Q)
	`t.*[2]`	$(\top,\top)$
$<A,A,A,B,C,C>$	`t.*[1]`	(A,B,C)
	`t.*[2]`	$(3,1,2)$
$(A,\{P,Q\},C,D)$	`t..q`	$((Q,\top))$
	`t..q[1]`	(Q)
$(A,(A,(A,B),C),D)$	`t..[1]`	(A,A,A)
	`t..[*]`	(A,A,A,B,C,D)

† Assuming t is represented as depicted in Figure 2.5 (p.54).

we discuss in the next bullet) and on how the result of each parallel task is combined with the results of others. In this setting, each $f_i(d_i)$ task of an embarrassingly parallel function can be thought of as returning a singleton set with the final result being arrived at by computing $e = f_1(d_1) \cup \cdots \cup f_m(d_m)$. Set union, $\cup$, is both associative and commutative, which means that partial results from different tasks may be incrementally assembled in any order into progressively larger subsets of e until the complete result e is acheived. So, not only is the computation of each $f_i(d_i)$ task distributively parallelisable but the incremental combination of partial results into a final single result is too.

2. **Memory usage** – Only one d_i element of d at a time needs to be loaded into working memory to compute one element $e_i = f_i(d_i)$ of $e = f(d)$. If we assume that individual elements of d and e can be separately read from and written to store then the computation of e does not require the entire d or e to be held in working memory. The maximum amount of working memory required at any one point is that required to to compute $e_i = f_i(d_i)$ in the worst case, which depends on the data and on f_i but is constant for a given computation of f. This means that the maximum working memory required remains constant and does not increase as the size of d and e increases, therefore enabling memory scalability in the size of inputs and outputs limited only by the size of store (assuming the maximum working memory required by any single f_i computation is available). This property holds whether the computation is carried out in parallel, under a shared memory architecture, or serially. In the case of distributed parallel execution the scalability is more facetted, benefiting from requiring only one d_i to be copied to each machine but suffering from having to copy (in a naive implementation) progressively larger subsets of e from machine to machine during the combination phase.

The above is only a sketch of the issues surrounding the parallelisability and scalability of embarrassingly parallel computation and hints at some of the complexities involved. In practice, the translation of an embarressingly parallel function into an efficient parallel and/or scalable implementation depends on numerous considerations that fall outside the scope of this Thesis. Recent texts from the fields of parallel and distributed computing give a more detailed introduction to the practicalities of Big Data systems design [Cou13, EPM13, CDKB11, RU11, Whi09].

We now consider the conditions under which *transformation* in the higher-order dataflow model introduced in this Thesis is embarrassingly parallel. Recall from Definition 4.3.2 (p.108) that the inputs $t_1, \ldots, t_n$ and output v of transformation $v = \Phi(g)(h)(t_1, \ldots, t_n)$ are basic relations with type $\mathfrak{R}$. Any basic relation $r \in \mathfrak{R}$ consists a pair (p, c), where basic term $p \in \mathfrak{B}$ represents properties of the relation and basic abstraction $c \in \mathfrak{B}_{\beta \to \gamma}$ represents a collection of items (our 'individuals') as depicted in Figure 4.7 (p.107). Intuitively, for a transformation with output relation $v = (p, c)$, if the computation of p does not depend on the computation of c then there exists a direct correspondence between the collection c and the output set e of an embarrassingly parallel computation $e = f(d)$. In other words, where c plays no part in the computation of p then each item $c_i \in c$ corresponds to each item $e_i \in e$ and each item may be computed independently in parallel.

Under these circumstances, the decomposed functions f_i that compute each e_i

all correspond to the single function ϕ_{Item} that computes each c_i in the transformation. This correspondence, without a slight extension, does not address relation property p. However, the computation of p by the function $\phi_{Relation}$ under these same circumstances can, from the perspective of an embarrassingly parallel function, be considered as corresponding to just another f_i function. So the f_i functions of an embarrassingly parallel function can all be mapped to just two functions in this particular kind of transformation: $\phi_{Relation}$ for the computation of p, and ϕ_{Item} for the computation of all $c_i \in c$. We refer to such transformations as *pure transformations* which are named after so-called *pure functions*. A pure function is the computer programming term for an implementation of a function such that it has no side effects[3]. The analogy is that in a pure transformation, the computation of c_i has no side effects on the computation of p. More formally we define pure transformation as follows.

Definition 4.3.16 (Pure Transformation) *For some* $t_1,\ldots,t_n \in \mathfrak{R}$ *and some* $v \in \mathfrak{R}$ *where* $t_1,\ldots,t_n = (p_1,c_1),\ldots,(p_n,c_n)$ *and* $v = (p,c)$, *a* pure transformation *is a transformation* $\Phi(g)(h)(t_1,\ldots,t_n) = v$ *where* p *has no dependencies on* $c_1,\ldots,c_n$ *such that* $\phi_{Relation}((p_1,c_1),\ldots,(p_n,c_n)) = p,\ \forall c_1,\ldots,\forall c_n \in \mathfrak{B}$.

The intended meaning of the above definition is that $\phi_{Relation}$ produces the same result p irrespective of the values of $c_1,\ldots,c_n$ and so depends only on the values of $p_1,\ldots,p_n$. Informally, $\phi_{Relation}$ behaves as if transformation were defined on $p_1,\ldots,p_n$ rather than on $t_1,\ldots,t_n = (p_1,c_1),\ldots,(p_n,c_n)$, i.e. as if defined as $\phi_{Relation}(p_1,\ldots,p_n)$ rather than $\phi_{Relation}(t_1,\ldots,t_n)$.

By contrast, note that pure transformation does not require ϕ_{Item} to only depend on $c_1,\ldots,c_n$, it may also depend on $p_1,\ldots,p_n$, and so continues to behave as $\phi_{Item}((c_1,\ldots,c_n),(p_1,\ldots,p_n))$, exactly as in ordinary (not pure) transformation. The reason for permitting ϕ_{Item} to have dependencies on the relation properties is that this does not affect the suitability of ϕ_{Item} as a decomposed function corresponding to $f_1,\ldots,f_n$ in an embarrassingly parallel function. If, however, $\phi_{Relation}$ in a pure transformation were permitted to depend on $c_1,\ldots,c_n$ then $\phi_{Relation}$ would be a poor choice as a decomposed function f_{m+1} because it could potentially access every item in every relation. While $\phi_{Relation}$ too could perhaps be decomposed there is, by design, nothing in the model to guide this decomposition, as imposing such restrictions on $\phi_{Relation}$ would remove the flexibility of the model – particularly for "small data" applications where the freedom in the choice of $\phi_{Relation}$ can simplify the overall dataflow. The utility of this design feature in our application context is touched upon again in Section 4.4 Implementation.

The bijective correspondences between outputs of an embarrassingly parallel function and outputs of a pure transformation are shown in Figure 4.11, along with similar bijective correspondences between the multiple functions that compute the items in these outputs.

So far in our discussion we have discussed correspondences between outputs e and v from an embarrassingly parallel function $e = f(d)$ and pure transformation $v = \Phi(g)(h)(t_1,\ldots,t_n)$ respectively. We have also discussed correspondences between the multiple functions f_i and the pair of functions ϕ_{Item}, $\phi_{Relation}$. Now, to

[3]Implementations with side effects should not really be called *functions* in the mathematical sense but they are in popular use outside of the functional and logic programming communities.

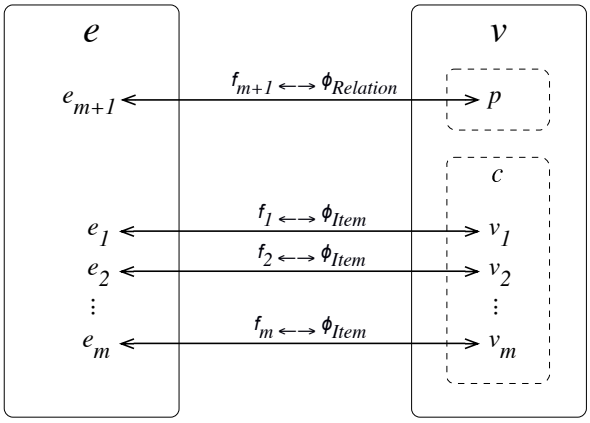


Figure 4.11: Correspondences between the output $e = \{e_1, \ldots, e_{m+1}\}$ of an embarrassingly parallel function and the output $v = (p, c)$ of a pure transformation, as well as respective correspondences between functions $f_1, \ldots, f_{m+1}$ and $\phi_{Relation}, \phi_{Item}$.

complete the picture we identify similar correspondences between the input d of an embarrassingly parallel function and inputs $t_1, \ldots, t_n$ of a transformation. To do so requires us to deal with the technical detail that a transformation can accept n input relations $t_1, \ldots, t_n$ rather than the single input set d, and that the structure of a relation is not identical to the structure of a set. These differences can be addressed by mapping the elements $d_i \in d$ to the corresponding elements in the set of n-tuples, $\{(p_1, \ldots, p_n)\} \cup \{(V(c_1\ b_1), \ldots, V(c_n\ b_n)) \ : \ b_1, \ldots, b_n \in \mathbb{N}\}$, where $t_1, \ldots, t_n = (p_1, c_1), \ldots, (p_n, c_n)$. In other words, there is a bijection between the set d and the set consisting of the union of the singleton n-tuple formed from all the n relation properties with set of the n-tuples formed from the b_i-th item of all the n relation collections. The indices b_i that determine which item of collection c_i appears at each position in the n-tuples depend of the choice of generator $g \in \{\rightarrow, \times, \lambda\}$ in the three cases of a transformation in Definition 4.3.2 (p.108).

We now more formally draw together the correspondences between the functions, inputs and output of pure transformation and an embarrassingly parallel function in the following proposition and proof.

Proposition 4.3.17 *Pure transformation is embarrassingly parallel.*

Proof 4.3.18 *To be shown is that there is an isomorphism between pure transformation $\Phi(g)(h)$ and embarrassingly parallel function f. This is true if there is an isomorphism between their signatures and if there is an isomorphism between their definitions. Assume $n \in \mathbb{N}$. Assume $\mathfrak{R}_1, \ldots, \mathfrak{R}_{n+1}$ are the types of basic relations. Trivially, there is a bijection between the signature of higher-order transformation function $\Phi(g)(h) : \mathfrak{R}_1 \times \cdots \times \mathfrak{R}_n \rightarrow \mathfrak{R}_{n+1}$ and the signature of an embarrassing function $f : D \rightarrow E$ such that $\mathfrak{R}_1 \times \cdots \times \mathfrak{R}_n \leftrightarrow D$ and $\mathfrak{R}_{n+1} \leftrightarrow E$, so the signature of $\Phi(g)(h)$ is isomorphic with the signature of f. To show that there is an isomorphism between the definition of $\Phi(g)(h)$ and the definition of f, we show that there is a bijection between their respective outputs, a bijection between their respective decomposed functions, and a bijection between their respective inputs.*

1. *Assume* $v = (p,c)$ *is the output to a pure transformation* $\Phi(g)(h)$*, where* $v \in \mathfrak{B} \times \mathfrak{B}_{\beta\to\gamma}$ *and* c *is*

$$\lambda x.\, \textit{if } x = b_1 \textit{ then } v_1 \textit{ else } \ldots \textit{ if } x = b_m \textit{ then } v_m \textit{ else } v_0,$$

for some $b_1,\ldots,b_m \in \beta$*, some* $v_1,\ldots,v_m \in \gamma$*,* $m \in \mathbb{N}$ *and default term* $v_0 \in \gamma$*. Assume* $o = \{p,c_1,\ldots,c_m\}$ *is the set of outputs from pure transformation* $\Phi(g)(h)$*, where* $c_1 = (b_1,v_1),\ldots,c_m = (b_m,v_m)$*. Assume* $e = \{e_1,\ldots,e_{m+1}\}$*, is the output of embarrassingly parallel function* f*. There is a bijection between* o *and* e *such that* $p \leftrightarrow e_{m+1}$ *and* $c_1 \leftrightarrow e_1,\ldots,c_m \leftrightarrow e_m$*, so* o *is isomorphic with* e*.*

2. *Assume* $h \in \mathfrak{B}_H$ *such that* $h = (\tau_{Relation},\tau_{Item},\tau_{Lambda})$ *for some template terms* $\tau_{Relation},\tau_{Item},\tau_{Lambda} \in \mathfrak{S}^c$ *with associated functions* $\phi_{Relation},\phi_{Item},\phi_{Lambda}$*. Assume* $\varphi_1,\ldots,\varphi_{m+1}$ *are the decomposed functions of pure transformation* $\Phi(g)(h)$ *such that* $\varphi_{m+1} = \phi_{Relation}$ *and* $\varphi_1 = \phi_{Item},\ldots,\varphi_m = \phi_{Item}$*. Assume* $f_1,\ldots,f_{m+1}$ *are the decomposed functions of embarrassingly parallel function* f*. There is a bijection between* $\{\varphi_1,\ldots,\varphi_{m+1}\}$ *and* $\{f_1,\ldots,f_{m+1}\}$ *such that* $\phi_{Relation} \leftrightarrow f_{m+1}$ *and* $\phi_{Item} \leftrightarrow f_1,\ldots,\phi_{Item} \leftrightarrow f_m$*, so* $\{\varphi_1,\ldots,\varphi_{m+1}\}$ *is isomorphic with* $\{f_1,\ldots,f_{m+1}\}$*.*

3. *Assume* $t_1,\ldots,t_n = (p_1,c_1),\ldots,(p_n,c_n)$ *is the input to pure transformation* $\Phi(g)(h)$*, where* $t_1,\ldots,t_n \in \mathfrak{R}$ *and* $n \in \mathbb{N}$*. Assume generator* $g \in \mathfrak{B}_G$ *where* $g \in \{\to,\times,\lambda\}$*. Assume* $\iota = \{(p_1,\ldots,p_n)\} \cup \iota_c$ *where* ι_c *depends on* g *such that*

$$\iota_c = \begin{cases} \{\ (V(c_1\ k),\ldots,V(c_n\ k)) : \forall k \in supp(c_1)\ \} & \textit{if } g = \to, \\ \{\ (V(c_1\ b_1),\ldots,V(c_n\ b_n)) : \\ \quad \forall b_1 \in supp(c_1),\ldots,\forall b_n \in supp(c_n)\ \} & \textit{if } g = \times, \\ \{\ (x,V(c_1\ b),\ldots,V(c_n\ b)) : \forall b \in supp(c_1), \\ \quad \forall (k,x) \in \phi_{Lambda}((V(c_1\ b),\ldots,V(c_n\ b)),(p_1,\ldots,p_n))\ \} & \textit{if } g = \lambda. \end{cases}$$

Assume $\iota = \{i_1,\ldots,i_m\}$ *is the set of inputs to pure transformation* $\Phi(g)(h)$*, where* $m \in \mathbb{N}$*. Assume* $d = \{d_1,\ldots,d_{m+1}\}$*, is the input to an embarrassingly parallel function* f*. There is a bijection between* ι *and* d *such that* $(p_1,\ldots,p_n) \leftrightarrow d_{m+1}$ *and* $i_1 \leftrightarrow d_1,\ldots,i_m \leftrightarrow d_m$*, so* ι *is isomorphic with* d*.*

As mentioned previously, proof that pure transformation is embarrassingly parallel does not guarantee than an implementation of our higher-order dataflow model will be highly parallelisable but it does strongly suggest that parallel implementation of the model is theoretically possible. Neither does the proof guarantee that an implementation will he highly scalable in the size of data that can be transformed but again it does suggest that scalable implementation is theoretically possible. More concrete support for the scalability of the model comes from our proof of concept implementation described in Section 4.4 Implementation. Our implementation demonstrates that pure transformations can be computed one result item $V(c\ k)$ at a time such that only the inputs to ϕ_{Item} (i.e. elements in ι from part three of Proof 4.3.18) need to

be loaded into working memory at once. This shows that working memory requirements for pure transformation can indeed be made independent of the number of items in input and output relations. There are other implications of this one-at-a-time computation of output items. These are the subject of the next section.

4.3.4 Continuations in the Model

A useful consequence of the embarrassingly parallel nature of pure transformation is that, if executed serially, the model requires very little state information in order to record the current program control state during the execution of a pure transformation. In practical terms this means that the iteration over the items in input relations may be suspended and its current control state saved to an abstract data structure, known as a *continuation*, which may be reloaded at a later time to resume the iteration from the point where it was suspended. Using such continuations enables suspension of the computation of a pure transformation, if required, inbetween every evaluation of the output item generator function ϕ_{Item} .

Before explaining the details of continuations in the higher-order dataflow model, we will first point out why they are useful in our web application use cases, where there is a trade-off between the need for responsiveness (returning results quickly) and scalability to larger data (requiring more time to process). As our implementation demonstrates, continuations make it possible to implement the model on a web server in such a way that complete results are returned quickly for small data and, for larger data, first results are returned quickly – leaving the computation of the rest of the results to proceed in the background over (potentially) a long period of time, guaranteeing that complete results will be available eventually. Our web service implementation achieves this by checking the elapsed time inbetween every execution of ϕ_{Item} and suspending execution within an acceptable response time, saving the state as a continuation; execution then moves to a background task that repeatedly restores the control state from the continuation, executes for a preset time, checking the time between each ϕ_{Item} and suspending control state as a continuation again when out of time. The intuition for this time-sliced serial execution of a set of discrete parallelisable tasks is depicted in Figure 4.12, which presents another way to think of this type of execution: as time distributed rather than processor distributed.

Of course, a wide variety of different architectures could be employed to achieve the responsive behaviour required in our web application use cases. For instance, scaling out to multiple compute nodes for Big Data computations involving large volumes of data. However, our implementation demonstrates that even on a modest single-node web server it is possible to process data that greatly exceeds the capacity of working memory without resorting to more complicated architecture. The pros and cons of this solution are discussed later in this chapter.

We now return to explain the details of continuations in the higher-order dataflow model. Provided that there exists an index of the keys to items in c_i for each input relation $t_i = (p_i, c_i)$, $t_i \in \Re$, and that each item in c_i is individually addressable then the state of the computation can be recorded by storing the current position within the index for each input relation, as shown in Example 4.3.19.

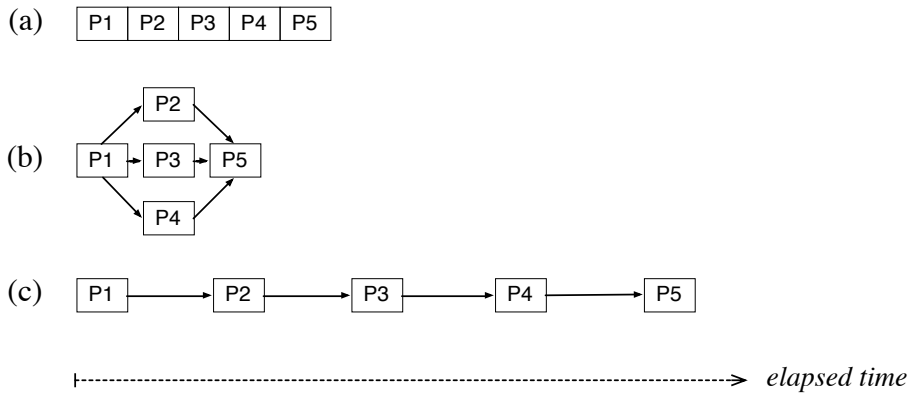


Figure 4.12: An illustrative toy 5-part computation, with parts P2-P3 parallelisable, executed as: (a) undistributed; (a) node distributed; (c) time distributed.

Example 4.3.19 *For a pure tranformation* $v = \Phi(\times)(h)(s,t)$, *for some* $h \in \mathfrak{B}_H$, *where input relation* $s = (p_s, \{(65,A),(66,B),(67,C)\})$ *and* $t = (p_t, \{(P,8),(Q,6)\})$, *for some* $p_s, p_t \in \mathfrak{B}$ *and some types* $\mathrm{M},\mathrm{N} \in \mathfrak{B}$ *such that* $A,B,C : \mathrm{M}$ *and* $P,Q : \mathrm{N}$, *the indexes will be* $[65,66,67]$ *and* $[P,Q]$ *respectively. In this example, the current control state within the computation can be identified by two counters* $j_s \in \{1,2,3\}$ *and* $j_t \in \{1,2\}$ *respectively (equivalent to two nested for loop counters) that identify an item from each index. The continuation is therefore representated by the pair* (j_s, j_t). *For instance, a continuation of* $(2,1)$ *identifies* 66 *and* P *from their respective indexes, which results in the individual computation* $\phi_{Item}(((66,B),(P,8)),\ (p_s,p_t))$. *If the next iteration increments* j_t *and computation were suspended at that point then the corresponding continuation would be* $(2,2)$. *Following that pattern, the full sequence of continuation pairs would be* $(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)$.

Implicit in Example 4.3.19 and in our overall description of continuations in model is that the current state of output relation $v \in \mathfrak{R}$ persists unchanged in store while the computation is suspended and that the original arguments $h \in \mathfrak{B}_H$ and $s,t \in \mathfrak{R}$ also similarly persist. To ensure this is the case requires that an implementation of the model could use a locking scheme that prevents changes to locked relations. Another strategy, avoiding this complexity for the model implementor, is to delegate the responsibility for not changing the input and output relations during an ongoing transformation to the user (web application programmer). An advantage of leaving this responsibility to the the programmer is that they are free to update parts of relations that have no involvement in the ongoing transformation. The obvious corollorary is that the complexity of understanding and ensuring correct behaviour rests with the programmer.

In the next section we introduce our own implementation of the higher-order dataflow model.

4.4 Implementation

The dataflow model and choice of representation introduced in this chapter were inspired by our earlier work on web services for profiling and matching text. Similarly, our implementation, called JSONMatch, is a generalisation of our earlier SubSift web services framework [FSG+09, PFS10b, PFS+13]. JSONMatch is hosted as a freely available resource by the University of Bristol. Further details are avilable on the JSONMatch website[4], along with extensive documentation of the API. The JSONMatch software is published under an open source licence and is available from the `jsonmatch` subversion repository on Google Code[5].

SubSift was designed to support workflows that computed the pairwise similarity between two sets of textual documents. JSONMatch is also able to perform this same task but instead of hard coding the workflow into the software, it is defined dynamically by supplying the higher-order parameters g and h to the dataflow transformations Φ via JSONMatch's REST API. This development and the adoption of basic terms as the knowledge formalism opens up new possibilities for defining different similarity measures and entirely different workflows, e.g. for cleaning and restructuring data, working with non-textual data, and incorporating external web services into dataflows.

4.4.1 Data Formats

JSONMatch, despite its name, is able to read, process and output a range of different data formats through its REST API, including plain text, CSV, XML, YAML, (Weka) AARF and, of course, its default of JSON. Appendix A.1 (p.223) includes a primer on the JSON data interchange format, for readers unfamiliar with JSON. Although JSON does not natively support basic terms, the following scheme intuitively affords their representation.

- *Basic Structures* are represented as JSON arrays, where the first element is the constant name. e.g. List $[A,B,C]$ represented as right-descending tree, $\#(A,\#(B,\#(C,[\,])))$, becomes `["#",A,["#",B,["#",C,"[]"]]]`. Of course, lists may also be conveniently represented as native JSON arrays.

- *Basic Abstractions* are represented as JSON objects, where each object key $k = \text{uniq}_\beta(s), s \in \text{supp}(t)$, for term $t \in \mathfrak{B}_{\beta\to\gamma}$, and each object value is a two-element array containing JSON representations of the key and value. The default term is implicit and not encoded in the JSON. e.g. An abstraction $\lambda x.$ if $x = (3,7)$ then 101 else if $x = (2.5,8)$ then 102 else ε, with $\text{uniq}_\beta : \beta \to \mathit{String}$ and $\beta = \mathit{Real} \times \mathit{Nat}$, in JSON becomes `{"(3,7)":[[3,7],101],"(2.5,8)":[[2.5,8],102]}`.

  Alternatively, if there is a suitable function uniq_β^{-1} such that $k = \text{uniq}_\beta(s) \wedge \text{uniq}_\beta^{-1}(k) = s, \forall s \in \beta$ then, the object values are simply the JSON representations of s. This closely matches the original basic abstraction. e.g. A set

[4] JSONMatch website – `http://jsonmatch.ilrt.bris.ac.uk`, visited May 2014.
[5] JSONMatch source code – `http://code.google.com/p/jsonmatch/`, visited May 2014.

{"*A*","*B*","*C*"} becomes `{"A":true, "B":true, "C":true}`. Similarly, a multiset $\{0,0,42,42,42\}$ becomes `{"0":2, "42":3}`.

- *Basic Tuples* are represented as JSON arrays. e.g. ("*A*","*B*",3.14,42) becomes `["A","B",3.14,42]`.

- *Types* are either represented implicitly by their JSON primitive types {`string`, `number`, `object`, `array`, `true`, `false`, `null`}, or explicitly by pairing each value with a string of its type name. e.g. A geolocation $(51.46, 2.6)$ of type $Geo = Lat \times Long$ becomes the JSON array `[[[51.46,"Lat"], [2.6,"Long"]], "Geo"]` or, if every type name in a value is known to be unique, the JSON object `{"Geo":{"Lat":51.46, "Long":2.6}}`.

4.4.2 Templates and Embedded Functions

An extension of the above scheme for basic terms also enables JSON representation of the higher-order template terms, $\tau_{Relation}, \tau_{Item}, \tau_{Lambda} \in \mathfrak{S}^c$, which may include embedded function applications. These templates are the JSON analogue of HTML templates used in web frameworks. Like HTML templates, JSON templates may include embedded function applications. Embedded functions are represented as JSON arrays, where the first element is the name of the function, as a string, and subsequent elements are the arguments of the function. JSONMatch has a library of built-in functions of the form $\mathfrak{B} \to \cdots \to \mathfrak{B} \to \mathfrak{B}$. Built-in functions all have names beginning `"jm:"`, designating the JSONMatch namespace. A trivial example is the function `jm:length`, which maps a string to its length, e.g. `["jm:length", "abc"]` will be substituted with `3`. Thus if embedded in the JSON object `{"x":"abc", "y":["jm:length", "abc"]}` in some template term, will result in `{"x":"abc", "y":3}`. Table 4.3 lists an illustrative selection of other built-in functions.

Thanks to an abbreviated form, one of the most important built-in functions, `jm:path`, rarely appears explicitly. `jm:path` applies a path expression to a basic term and has the definition $\text{path}(e,t) = t|_e$, where $t \in \mathfrak{B}$ and $t \in \mathcal{P}_e(t)$. Path expressions in JSONMatch are implemented as JSONPath expressions[Gös07], but without JSONPath's web-unsafe evaluation of arbitrary functions in the underlying language. JSONPath for JSON is similar to the better known XPath for XML but is substantially simpler to learn and use. JSONPath, and JSONMatch by implication, admits syntactic variants other than the path expressions of our model but shares the same semantics. Appendix A (p.223) is a primer on JSONPath for readers unfamiliar with the language. The primer includes links to implementations in a variety of programming languages and to an interactive online expression tester, the latter being a convenient way to try out JSONPath queries without having to write any code.

The most common usage of path expressions in our model is to access subterms of the inputs to ϕ_{Item} when computing each item $V(c\ k)$ in the output relation of a transformation. These inputs are available to path expressions via an array, `items`, in an object called `context`. For an n-ary transformation, `items` will have n elements: each one an item from the transformation's respective n input relations. As syntactic sugar, JSONPath strings in template terms are implicitly treated as

Table 4.3: EXAMPLE JSONMATCH FUNCTIONS

Function	Description
`jm:set`	Creates a set from an array/list.
`jm:set_union`	Union of two sets.
`jm:multiset`	Creates a multiset from an array/list.
`jm:multiset_union`	Sums the multiplicities of the union.
`jm:apply`	Apply function to each element of a list.
`jm:project`	Extract subterm from a basic term.
`jm:types`	Detect types of all subparts of a JSON value.
`jm:http_get`	Fetches text of web page; or REST call.
`jm:extract`	Substring matching a regular expression.
`jm:remove_html`	Strips HTML mark-up from a string.
`jm:downcase`	Converts a string to lower case.
`jm:split`	Splits a string into a list of words.
`jm:entities`	Recognises names of people, places, etc.
`jm:stem`	Replaces words with their Porter stem.
`jm:ngrams`	Returns *n*-grams from a list of words.
`jm:pgrams`	Returns *p*-spectrum of a list of words.
`jm:prefixes`	Counts left substrings of a list of words.
`jm:exclude_words`	Removes supplied 'stopwords' from list.
`jm:include_words`	Restricts words to supplied vocabulary.
`jm:replace_words`	Substitutes words using supplied mapping.
`jm:tfidf`	Calculates TF-IDF for a multiset of words.
`jm:cosine`	Cosine similarity of term-weight vectors.
`jm:kernel`	Kernel (as a distance) on JSON values.

path queries on the `context` object, e.g. `"$.items[0].author.name"` is treated as `["jm:path", "$.items[0].author.name", context]`.

4.4.3 Example Dataflows

The dataflow examples in this section consist of a series of HTTP method calls to the JSONMatch REST API and follow the same concise notation used in our SubSift REST API workflow enactment examples from Chapter 3 (see Section 3.2.1, p.82). To recap, HTTP request methods and their parameters are denoted using the format below and no HTTP responses are shown. We omit `<uri>`, which has the value `http://jsonmatch.ilrt.bris.ac.uk` for the publicly hosted version of JSONMatch, and `<user_id>`, the account name (e.g. `ecmlpkdd12` for the ECML-PKDD'12 conference).

```
<http_method> [<uri>][<user_id>]<path>
<parameter_name_1> = <parameter_value_1>
<parameter_name_2> = <parameter_value_2>
...
<parameter_name_N> = <parameter_value_N>
```

As before we also omit details of the security token needed in the HTTP request header of all `DELETE`, `POST` and `PUT` requests. The token is also required to access relations and data marked as private, irrespective of request method.

Our first dataflow example is motivated by the real-world task of harvesting and profiling the DBLP Computer Science bibliography author pages of Programme Committee (PC) members for the ECML-PKDD 2012 conference. This corresponds to the PC profiling part of the submission sifting workflow, highlighted in Figure 4.13 which depicts a composition of bookmark, document and profile service components working in conjunction with a web harvester robot. In the following example, we replace these separate service components (as well as the robot) with repeated uses of a single JSONMatch transform component with each use controlled by higher-order parameters, as depicted in Figure 4.14. The transform service component is an implementation of transformation $\Phi(g)(h)$ from our higher-order dataflow model. In JSONMatch the higher-order parameters to this component are named `generator` and `templates`, corresponding to g and h respectively, and are passed in as HTTP parameters. The `generator` parameter is one of `map`, `product`, `lambda` representing g values of $\rightarrow$, $\times$, λ respectively. The `templates` parameter is a JSON object representing higher-order templates $\tau_{Relation}$, τ_{Item}, τ_{Lambda} in the form of arbitrary JSON data (possibly containing embedded functions), associated with the keys `"relation"`, `"item"`, `"lambda"` respectively. Below we describe each of the four transformations in the sequence from Figure 4.14 that produces the PC profiles.

Transformation 1 (pc ← CSV)**.** The input to the dataflow is a CSV file with rows: `<pc_member_name>`, `<url>`, `<primary_keyword>`, `<keywords>`. This can be loaded into JSONMatch to create a basic relation called `pc`, with `pc_member_name` as the item id as specified by the JSONPath expression `"$[0]"`, in the following request.

```
POST /relations/pc/items
from=csv
id_path=$[0]
value=<text of csv file>
```

The resultant basic relation `pc` has type $\mathfrak{R}_{Id \rightarrow Name \times Url \times Primary \times Keywords}$, where $Id, Name, Url, Primary, Keywords \in \mathfrak{B}_{String}$. This means that the individual items in the basic abstraction (i.e. collection) part of basic relation `pc` have type $Id \rightarrow Name \times Url \times Primary \times Keywords$ and constitute the *individuals as terms* referred to in "Section 2.3.2 – Individuals as Terms" (p.50). All the information about a single PC member is represented in a single basic term, a basic *4*-tuple in this case. So, in the following JSON representation of basic relation `pc` the JSON object associated with the key `item` contains the individual PC members keyed on their *id* and formatted as an array of strings. The example has been abbreviated for readability, showing

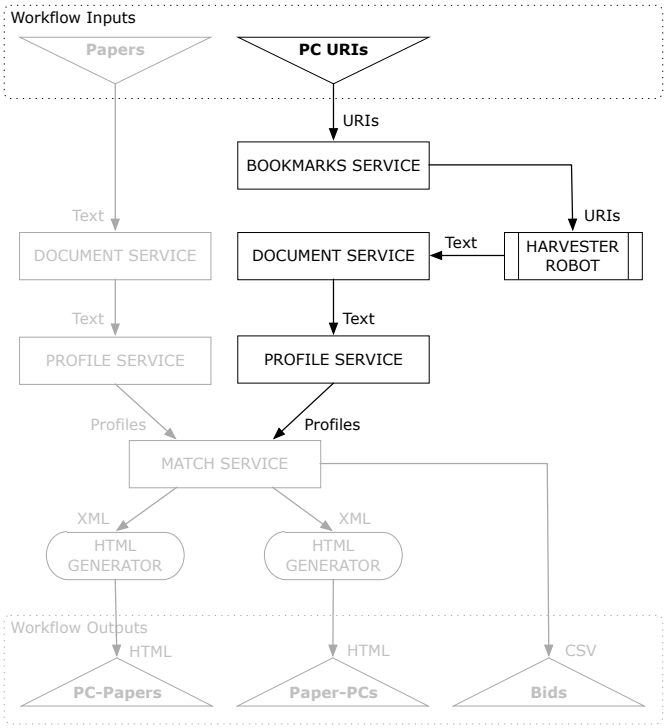


Figure 4.13: Schematic highlighting PC profiling part of *submission sifting* workflow.

only the first three PC members and only one (i.e. Ad Feelders) in approaching full detail.

```
[
  "relation": {
    "id": "pc"
  },
  "item": {
    "Ad_Feelders": [
      "Ad Feelders",
      "http://dblp.uni-trier.de/pers/hd/f/Feelders:Ad.html",
      "Rankings_and_Partial_Orders",
      "Active_Learning; Bioinformatics; Constraints; ..."
    ],
    "Aditya_Menon": [
      ...
    ],
    "Alain_Rakotomamonjy": [
      ...
    ],
    ...
  }
]
```

No explicit relation properties were specified in this initial transformation of CSV data to basic relation `pc` and so the `relation` properties of `pc` will only contain some

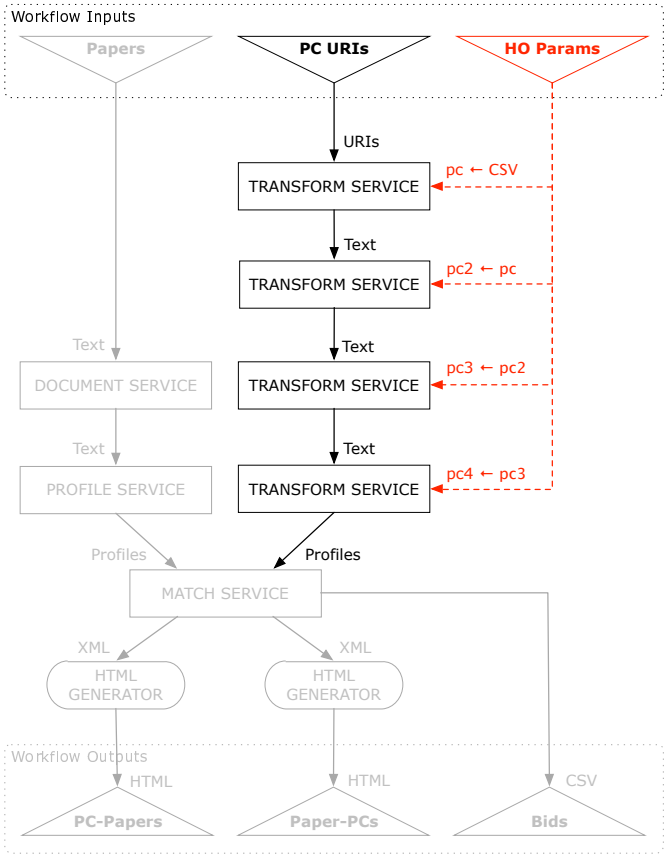


Figure 4.14: JSONMatch implementation of PC profiling part of *submission sifting* workflow. Four sets of higher-order parameters ("HO Params") control the behaviour of the four invocations of the transform service component. The first set "pc ← CSV" defines a transformation from CSV data (i.e. $\mathfrak{R}_{\alpha_1 \times \cdots \times \alpha_n}$ for types $\alpha_1, \ldots, \alpha_n$) to basic relation $\mathfrak{R}_{\text{pc}}$, the second set "pc2 ← pc" defines a transformation from $\mathfrak{R}_{\text{pc}}$ to basic relation $\mathfrak{R}_{\text{pc2}}$, and so on.

automatically generated metadata, such as the `id` of the relation and provenance information about the transformation plus some access permissions information. Relation metadata apart from `id` is not shown in these examples. However, the metadata fields are shown in the API documentation on the JSONMatch website[6].

Transformation 2 (pc2 ← pc)**.** A map transformation now converts the *4*-tuple representing each individual in relation `pc` to an object with five key-value pairs in relation `pc2` using the HTTP request shown below.

```
POST /relations/pc2/from/pc
generator=map
templates={
  "item": {
    "name": "$.items[0][0]",
    "url": "$.items[0][1]",
    "primary": "$.items[0][2]",
    "keywords": ["jm:split", "$.items[0][3]", {"regex": ";"}],
    "text": ["jm:http_get", "$.items[0][1]"]
  }
}
```

Three of the five key-value pairs use JSONPath expressions to select one of the four elements of the *4*-tuple, e.g. the expression `"$.items[0][1]"` evaluates to the second element of each tuple (from the first input relation `"$.items[0]"`, i.e. `pc`), which is the url of the PC member's DBLP author page. The keywords value, which was a semicolon-delimited string in the original csv data, is split into an array of separate strings using the embedded function `jm:split` applied to the fourth element of the input tuple and a trivial regular expression to locate each occurrence of the delimiter. A fifth key-value pair is added to each individual object such that the key is `text` and the value is the HTML fetched from each PC member's DBLP author page by embedded function `jm:http_get`, which takes a url as its single argument.

After the transformation, each individual item in `pc2` has the following form.

```
"Ad_Feelders": {
  "name": "Ad Feelders",
  "url": "http://dblp.uni-trier.de/pers/hd/f/Feelders:Ad.html",
  "primary": "Rankings_and_Partial_Orders",
  "keywords": [
    "Active_Learning",
    "Bioinformatics",
    "Classification",
    ...
  ],
  "text": "<html><head><title>A. J. Feelders</title>...</html>"
}
```

Transformation 3 (pc3 ← pc2)**.** The next map transformation constructs a multiset of word (term) frequencies in the string obtained by stripping HTML mark-up from

[6] API documentation – `http://jsonmatch.ilrt.bris.ac.uk/api`, visited April 2014.

the text. For clarity we have omitted the additional trivial functions calls to append all the keywords to the end of the text, convert the text to lower case, and append 2 -grams and 3 -grams of all the words to the list prior to counting frequencies.

```
POST /relations/pc3/from/pc2
generator=map
templates={
  "relation": {
    "corpus_n": ["jm:multiset_union", "$.relation.item[*].term_n"],
    "corpus_dt": ["jm:multiset_increment", "$.relation.item[*].term_n"],
    "corpus_N": "$.arguments_size.[0]"
  },
  "item": {
    "name": "$.items[0].name",
    "url": "$.items[0].url",
    "primary": "$.items[0].primary",
    "keywords": "$.items[0].keywords",
    "term_n": ["jm:multiset",
      ["jm:split", ["jm:remove_html", "$.items[0].text"]]
    ],
    "document_n": ["jm:multiset_cardinality", "$.item.term_n"]
  }
}
```

Some explanation of the `relation` template term is required. The functions with dependencies on path expressions involving `$.relation.item[*]` are compiled by JSONMatch to evaluate after each item's template term is evaluated. In effect, they serve as global accumulators and are a convenient shortcut to avoid the more parallelisable MapReduce approach to counting. To achieve scalability to large data, this shortcut cannot be used.

Transformation 4 (pc4 ← pc3)**.** A final map transformation then profiles each author by adding a `term` (word/phrase), a list of term to TF-IDF pairs, which is obtained by computing the TF-IDF weighted score for the `term_n` multiset of term frequencies. Optional extra functions can be used to sort, threshold and limit the number of pairs in the result.

```
POST /relations/pc4/from/pc3
generator=product
templates={
  "relation": {
    "corpus_n": ["jm:multiset_union",
     ["jm:project", "$[*][0,2]", "$.item.term"]],
    "corpus_dt": ["jm:multiset_increment",
     ["jm:project", "$[*][0]", "$.item.term"]]
  },
  "item": {
    "name": "$.items[0].name",
    "url": "$.items[0].url",
    "primary": "$.items[0].primary",
    "keywords": "$.items[0].keywords",
    "document_n": "$.items[0].document_n",
```

```
    "term": ["jm:tfidf",
      "$.items[0].term_n",
      "$.items[0].document_n",
      "$.relations[0].corpus_dt",
      "$.relations[0].corpus_N"]
  }
}
```

These profiles in the `item` list may then be used in a product transformation on a pair of profile relations to compute cosine similarities between the profile abstractions. The full sequence of REST API calls for a complete submission sifting dataflow is included in Appendix C (p.237).

4.4.4 Other Example Dataflows

In recreating and extending the functionality of SubSift in JSONMatch, lambda transformations proved exceptionally useful in data preparation and transfer between the systems - completely avoiding the need to write any migration code other than that embedded in the higher-order templates. For lambda transformations, the function `jm:generate`, having signature $\tau_{Lambda} \times List \rightarrow \mathfrak{B}$, outputs a term constructed by evaluating the template term for each element in the list. The template may itself contain embedded functions, evaluated against an implicit local context that includes `item`, the result of the application of a JSONPath expression, τ_{Item}, to the current global `context`. Other useful applications of lambda generators include splitting and grouping data. e.g. For a transformation on a relation containing RSS blog data,

```
{
  "item": ["jm:generate", "$.item",
           "$.items[0].rss.channel.item[*]"]
}
```

will emit a separate item for each channel item (i.e. story) in the RSS feed. Note that the `item` in `rss.channel.item[*]` is an element name from the RSS schema and is unrelated to the JSONMatch `item` value in the current `context`. Continuing our example, the list of channel items produced by the above transformation can then be grouped together by author, simply by generating into item 'bins' of the same id.

```
{
  "item": ["jm:generate",
     ["jm:push", "$.item", "$.old_item"],
     "$.items[0].rss.channel.item",
     {"id_path": "$.item.dc:creator",
      "unique_id": false, "fetch_item": true}
  ]
}
```

The function `jm:push` adds an item to the end of a list. Here the list is the previous value of the item already saved at this id, accessed through the local context `old_item`. The optional third argument to `jm:generate` contains `id_path` which specified the id of the item to be created (or updated), flags for whether that id should

be allowed to be re-used and whether to fetch the old value of the item and make it available as `old_item`. This idiom of merging or overwriting items already generated is both flexible and powerful. One useful application is to produce Web 2.0 AJAX type-ahead predictive text look-up web service by generating items whose ids are all the left substrings of a set of words, using `jm:prefixes`. Such functionality normally requires a special-purpose text engine such as Apache Lucene.

4.5 Comparison of JSONMatch with SubSift

In "Chapter 6 – Related Work" we compare JSONMatch to previous work elsewhere on dataflow systems. In this section we compare JSONMatch and SubSift, beginning with a discussion of their respective implementations of the *submission sifting* workflow, as originally depicted in the schematic from Figure 3.9 (p.87). In preparation for this discussion, we first recall details of the SubSift workflow steps implementing submission sifting and the equivalent (partial) implementation of the same workflow using JSONMatch[7]. We then develop the latter further into a full implementation of the submission sifting workflow, before concluding with presentation and discussion of a side-by-side feature comparison of the proof of concept frameworks themselves.

4.5.1 Submission Sifting Workflow Implementation

A sequence of ten REST API method calls implementing the submission sifting workflow's dataflow in SubSift was given in Section 3.2.1 (p.82). In the current chapter, our example JSONMatch dataflow from Section 4.4.3 (p.125) described a partial implementation of the same workflow, specifically the subpart profiling DBLP author pages of Programme Committee members. Figure 4.14 (p.128) highlighted the corresponding subpart of the workflow under consideration and we only gave example REST API method calls for that subpart. The fact that for brevity we chose not to give the complete sequence of JSONMatch REST API calls required to implement the full workflow gives a hint at one of the obvious differences between the two frameworks: JSONMatch calls typically require more information to be provided by the web application or workflow developer (user) and are therefore more verbose than their SubSift equivalents.

The main reason for requiring this extra information is that JSONMatch does not have built-in notions of relation types that correspond to SubSift's built-in `bookmarks`, `documents`, `profiles`, `matches` and `reports` folder types and built-in behaviours; instead, JSONMatch requires the user to supply a higher-order `templates` parameter embodying these concepts at enaction-time when each transformation method is invoked. On the one hand this gives the user more control over the dataflow transformations possible in JSONMatch, allowing transformations that were not hard-coded into JSONMatch itself; in SubSift the only transformations possible are those hard-coded by its developers. The downside of this additional power in JSONMatch is that the functionality of transformations must be specified in the REST API `templates` parameter – requiring the user to have both an understanding of the requisite embedded functions and, of course, to provide more parameter data in each method call.

[7] A complete implementation of the workflow in JSONMatch is given in Appendix C.

A second reason why JSONMatch dataflows may require more information to be provided by the user to implement equivalent dataflow to SubSift is that multiple JSONMatch transformations are sometimes required to implement the same functionality of a single SubSift method call. For example, SubSift's profiling method makes two complete passes over the data: the first pass to compute corpus totals (i.e. the total number of occurrences of each word over all documents in a folder and the number of documents in which that word occurred); the second pass to compute the TF-IDF scores of words in each document normalised using those corpus totals. JSONMatch requires two separate REST API method calls to implement the same functionality as two separate transformations. For this reason, JSONMatch requires the four transformations depicted in Figure 4.14 as opposed to SubSift's three.

Regardless of their respective number of REST API method calls required to implement submission sifting in JSONMatch versus SubSift, it should be noted that the overall volume of data transmitted over HTTP between client and server is broadly the same in both frameworks. The data volume transmitted is dominated by the original ingest (upload) of the data and the final download of the resulting profiles and matches data. All the intervening calls only involve the transmission of parameters, which trigger the frameworks to access their shared data stores.

Thus far we have only discussed a JSONMatch four-transformation partial implementation of the submission sifting workflow, corresponding to just three of the nine steps depicted in Figure 3.9 from the original SubSift implementation. In Figure 4.15 we replace the remaining six steps of the SubSift workflow with five additional JSONMatch transformations, plus two HTML generation steps, to implement the full workflow in JSONMatch. The HTML generator components are part of a separate application view layer and not of themselves part of JSONMatch. These generators are common to both SubSift and JSONMatch workflows, but in SubSift they are an integral part of the reporting methods whereas in JSONMatch this view layer is absent and any web templating system (e.g. Django, Rails or Template) can be used to render transformation outputs as HTML.

Figure 4.15 includes nine sets of higher-order (HO) parameters controlling the behaviour of the nine invocations of the transform service component. In JSONMatch, these parameters are represented as JSON with embedded `jm:` functions, and passed via the `templates` argument to REST API methods. Recall from the earlier partial example that higher-order parameters "pc ← CSV", "pc2 ← pc", "pc3 ← pc2" and "pc4 ← pc2" define a sequence of map transformations from Programme Committee DBLP author page URLs to PC member profiles. Here we replace a similar sequence of SubSift workflow steps with three further map transformations defined by higher-order parameters "a ← CSV", "a2 ← a" and "a3 ← a2" to produce profiles of the submitted papers (the "a" stands for abstracts). SubSift's match component is replaced in the JSONMatch workflow by a transformation with higher-order parameter "m ← a3 × pc4", the output of which is fed directly into HTML generators to produce reports and into a final map transformation, "bids ← m", to output initial bids based on the cosine similarity values in the match data. Notice that in JSONMatch, only one output is produced, compared to the three from SubSift's match component, which leaves additional work for the view layer in producing reports and initial bids data. The full sequence of REST API method calls required to produce this dataflow is included in Appendix C.

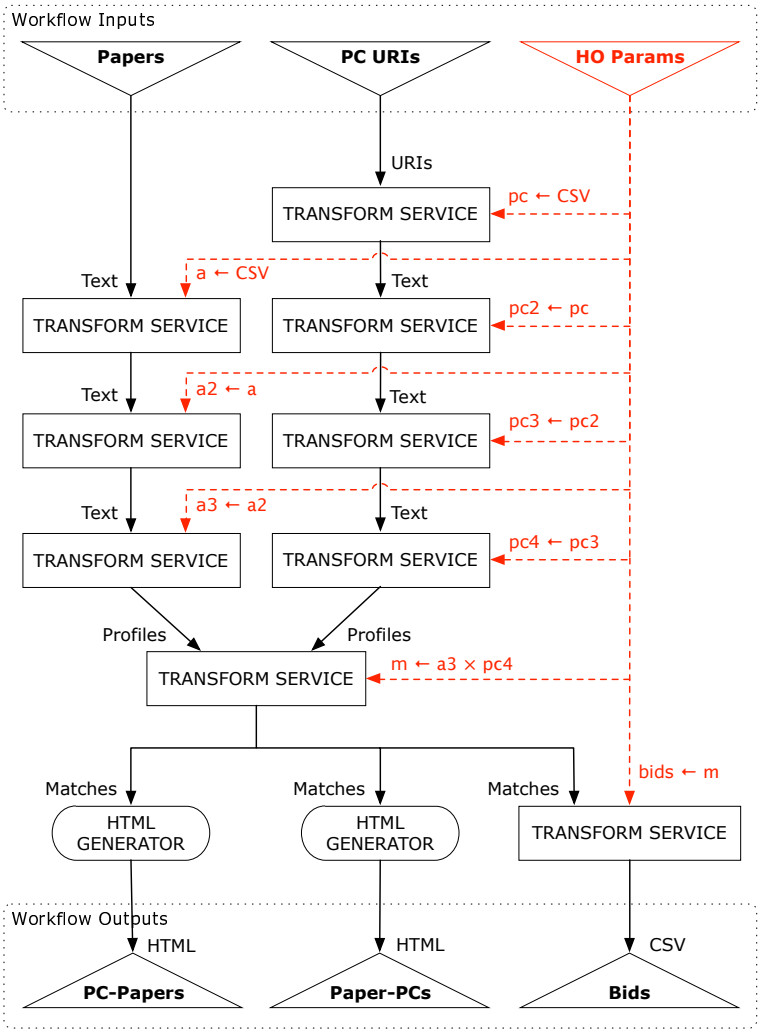


Figure 4.15: JSONMatch implementation of the complete *submission sifting* workflow, employing nine sets of higher-order parameters controlling the behaviour of the nine invocations of the transform service component.

One final observation about the JSONMatch implementation of the submission sifting workflow, compared to the SubSift implementation, is that they are structurally very similar if we ignore the previously discussed extra map transformations for multiple passes over the data. Typically, each JSONMatch transformation, plus its associated higher-order parameter, corresponds to a single SubSift component.

4.5.2 Asynchronous Behaviour

Interestingly, in this same dataflow from Figure 4.14, SubSift wraps its functionality for asynchronously fetching web pages (the DBLP author pages in this example) in a standalone web harvester robot. JSONMatch achieves similar asynchronous functionality through an embedded function call to `jm:http_get` inside the `item` template passed to the method call and therefore has no standalone web harvester *per se*. By

'asynchronous' here we mean that the REST API method will return quickly but that the process of harvesting the web pages will continue in the background, completing at some time in the future. JSONMatch supports this behaviour for all transformations such that if a transformation is not completed within the HTTP timeout period[8] then the task moves to the background; SubSift supports this asynchronous behaviour only when fetching web pages (by virtue of its separate web harvester) and so any other web timeouts will result in an HTTP error response from the server because SubSift, unlike JSONMatch, has no model for pausing and resuming an algorithm mid-way through. Both frameworks rely on the client polling the server, using lightweight HTTP requests, to detect when an asynchronous process has completed. The alternative of using web sockets[9], unlike HTTP itself, is not stateless and so is less scalable in the number of clients but would be more efficient in terms of reducing the number of interactions between client and server. We chose polling for its robust HTTP statelessness and ease of implementation in both server and client code for our proof of concept frameworks.

The aforementioned example series of JSONMatch method calls from Section 4.4.3 performs this HTTP GET step in a separate transformation to the pre-profile transformation (i.e. pass one over the data) to closely follow the behaviour of SubSift, but could alternatively have skipped this separate step and incorporated the HTTP GET directly into the pre-profile transformation instead. Whether this is desirable or not is application dependent, but does illustrate the extra control JSONMatch affords the user in exchange for the increased complexity of having to provide the higher-order templates. Again this flexibility comes at a cost of increased design cost for the end user, who must make such decisions when invoking JSONMatch's transformation API methods. In the Future Work section of Chapter 8 we return to this issue and suggest a way of separating out the template design task from template use, in effect creating different layers for different audiences.

4.5.3 Feature Comparison

In Table 4.4 we compare features of JSONMatch and SubSift side-by-side. As these are both proof of concept implementations it makes little sense to present detailed performance data; instead we highlight some interesting similarities and differences between the two systems in order to emphasise their relative advantages and disadvantages in different settings. The table groups features into four broad categories covering code, performance, input/output formats and REST API, each of which we discuss below.

Code

Considering code size first, SubSift is 30% larger (measured in lines of code) than JSONMatch but this difference is largely accounted for by the former's reporting features which are absent from JSONMatch. Module (third-party library) dependencies are roughly the same, with JSONMatch's storage layer accounting for its slight longer

[8] This is typically defined by a configuration parameter in Apache or another web container/server

[9] W3C web sockets – `http://www.w3.org/TR/websockets/`, visited April 2014.

Table 4.4: JSONMATCH AND SUBSIFT COMPARISON

Feature	**SubSift**	**JSONMatch**
Implementation language	perl	perl
Source files (perl)	68	46
Lines of code (perl)	12,117	8,060
Dependencies (cpan modules)	33	36
Storage layer	file system	BerkeleyDB
Record directory	file directories	SQLight
Largest computation size	$< 10^4$ items	$> 10^6$ items
Computation size constraint	working memory	storage or time
Working memory (profile)	linear in no. items	constant
Working memory (match)	quadratic in no. items	constant
Reads from store (profile)	N reads / N items	$2N$ reads / N items
Reads from store (match)	N reads / N items	$\frac{N^2}{2}$ reads / N items
Input formats	csv, text	arff, csv, json, rdf, text, xml
Default input format	csv / text	json / text
Output formats	csv, json, rdf, terms, xml, yaml	csv, json, rdf, terms, xml, yaml
Default output format	xml	json
Report formats	csv, dot, html, zip	*none*
API groups	14	3
API group names	`bookmarks +items`, `documents +items`, `profiles +items`, `matches +items +pairs +matrix`, `reports +files`, `workflow`, `system`	`relations +items`, `workflow`
API methods	78	21
API parameters	172	27
Embedded functions	0	53
Extension mechanism	edit source code	`templates` parameter, built-in/local/external embedded functions

list of dependencies. From the similar sizes and dependencies of both implementations it can be argued that the frameworks are of a broadly similar size, although it should be kept in mind that JSONMatch has abstracted a layer of complexity out of

the system and passed it on to the user – so in theory, considerable additional code may be incorporated into JSONMatch during workflow enactment by means of the `templates` parameter and its embedded functions. As we will describe below, these embedded functions enable users to incorporate their own bespoke functions into JSONMatch. SubSift has no counterpart to this mechanism and instead relies on the calling web application or the containing workflow system.

Performance

The performance characteristics listed in the table are deliberately high-level but have been chosen to illustrate that there is no free lunch in moving from SubSift to JSONMatch. SubSift uses in-memory computation and so is inherently faster than JSONMatch on computations where input/output time makes a significant contribution to the overall running time. In JSONMatch, cross product computations, such as the match step in submission sifting, are particularly expensive in terms of the number of times that items are read from store; by contrast, SubSift reads each item once and holds all items required for a computation in working memory, avoiding the need to re-read items multiple times as JSONMatch does. For cross product computations involving a total of N items, JSONMatch requires N^2 item reads, or $\frac{N^2}{2}$ item reads if the result is symmetric, as is the case with similarity, distances and kernels, e.g. in the submission sifting match step. However, the payback for JSONMatch having sacrificed the benefits of in-memory computation come from its ability to process data that exceeds the size of working memory, thereby enabling computations involving more items than it is possible to hold in working memory. This is a consequence of the scalability and parallisability of JSONMatch's underlying higher-order dataflow model, as discussed in Section 4.3.3.

Input/Output Formats

The input formats for SubSift are predominantly plain text, with the exception of bulk ingest API methods which read CSV files, but imported data is always restructured into a hard-coded data structure associated with the relevant folder type (i.e. `bookmarks`, `documents`, etc.). JSONMatch accepts a wider range of input formats either preserving the structure of the original document (e.g. an XML tree or a JSON object) or selecting subparts of the structure using JSONPath (e.g. to specify that a particular field should be used as the item identifier). However, both frameworks share the same range of output formats and differ only in their choice of default format. JSONMatch has no equivalent to SubSift's built-in reporting features and so, in this sense is a less self-contained system than SubSift. This choice was deliberate as the inability to fine-tune reports without modifying the reporting code itself was a frustration when using SubSift to support the ECML-PKDD 2012 conference; JSONMatch allows the user to fine-tune the data that feeds into a report through the use of higher-order transformations, the output of which can then be piped through a standard HTML templating system as previously described. Again, this flexibility is achieved at the cost of ease-of-use, a recurrent theme of this discussion which we pick up again in the Future Work section of Chapter 8.

REST API

The statistics of the two APIs might at a first glance appear to show that JSONMatch is simpler to learn and use than SubSift but, by this point in our comparison, it should now be clear that this is not the case: SubSift has a rich range of pre-defined methods that provide the user with a scaffold around which to more easily build applications that can be mapped onto the SubSift profile-match model; JSONMatch has fewer API methods but those relating to higher-order transformations abstract complexity out to libraries of embedded functions, providing the user with a more adaptable and extensible model than SubSift. Appendix B (p.229) further describes the range of built-in embedded functions available in JSONMatch as well as documenting internal and external mechanisms for extending the range of functions available.

4.5.4 Summary of Comparision

We have shown that JSONMatch is capable of implementing the submission sifting workflow but that this requires more steps in the dataflow and more information to be supplied by the user. The benefit of this extra user effort is greater control over the process and, we suggest, increased flexibility in adapting this canonical workflow to other use cases. JSONMatch also frees submission sifting computations from the size constraints of SubSift's in-memory computation (at the cost of reduced cross product speed in matching), although this is not a constraint arising from SubSift's model itself – only from our current implementation. In principle, it is possible to rewrite each of SubSift's methods to support the processing of larger datasets in order to obtain similar performance characteristics to JSONMatch – without needing to change SubSift's hard-coded profile-match model. Additional input and output format support could also be added to SubSift retrospectively, with the contraint that all data must be restructured internally into SubSift's pre-defined data types; the two frameworks are somewhat different in this regard. However, the major, indeed fundamental, difference between SubSift and JSONMatch is the latter's use of a higher-order `templates` parameter with embedded functions. This affords the user a far more flexible and powerful software engineering tool, but at the cost of increased application design effort. Learning to use JSONMatch requires the user to understand map, product and lambda transformations, JSONPath, data structures and the range of built-in embedded functions as well as, possibly, how to extend the range of functions available. In short, a JSONMatch user has to learn to be a "JSONMatch programmer" rather than just an API user.

4.6 Summary

In this chapter we explored a further abstraction of submission sifting that led us to propose a higher-order dataflow model that is intended to be more flexible and generalised than the SubSift model. The model ranges over a class of terms in a strongly typed higher-order logic that have previously been shown to be sufficiently expressive to represent a wide variety of unstructured, semi-structured and structured data [Llo03, GLF04]. We proved that a certain class of transformations in the model, which we refer to as *pure transformations*, satisfy the conditions of an *embar-*

rassingly parallel function and hence, in principle at least, are highly parallelisable. JSONMatch, our proof of concept web services implementation of the model demonstrates that working memory requirements for pure transformation can in practice be made independent of the number of items in input and output relations, supporting our claim that pure transformations are, in principle, highly scalable.

Later in "Section 6.3.2 - Comparison with Other Software Frameworks" (p.194) we compare JSONMatch to previous work on dataflow systems but in the current chapter we limited our comparisons to our own SubSift framework. As part of our comparison with SubSift we demonstrated that JSONMatch has the necessary expressive power to be able to implement submission sifting workflows through flexible user-supplied higher-order parameters to a dataflow at runtime, rather than hard-coding this specific workflow pattern into the framework itself. We also observed that the increased flexibility of JSONMatch comes at the cost of a more expensive design process for users (developers), as compared to SubSift, because much more of the ultimate functionality is left to the user to design. In the Future Work chapter of this Thesis we revisit this issue and propose a scheme for adding a further API wrapper to JSONMatch so that different levels of user would access the functionality in different ways, in effect allowing the creation of SubSift-like packaged problem-specific solutions in JSONMatch.

Chapter 5

Querying and Merging Heterogeneous Data

In Chapter 2 we described the multifarious problems encountered when representing heterogeneous structured data in relational representations but, at the time, we also remarked that such representations are still widely used in practice. This is because the representational difficulties are outweighed in many application domains by other advantages of the relational model, particularly the convenience of the SQL manifestation of the relational algebra. Relational databases and SQL form an important element of many workflow systems, not only as part of their implementation architecture but also as a parameter in their web service APIs. For instance, Example 5.0.1 describes just one such SOAP service listed in the BioCatalogue life science web service directory[1]. Explicit SQL parameters are less common in REST than in SOAP because the URLs of the former often implicitly represent SQL queries, such as the example given in "Section 2.3.5 – RESTful Web Services" (p.61). Consequently the query formalism is even more important within REST because the URL design tends to directly map to queries.

Example 5.0.1 (Nuclear Protein Database Query Service) *The Medical Research Council's Human Genetics Unit maintains a searchable Nuclear Protein Database which has both human and web service interfaces. Details of proteins may be retrieved by name, motif, compartment or keyword by submitting SQL queries via their SOAP API method,* `SQLquery`*, the documentation of which we reproduce below.*

```
SQLquery(string SQLquery) - output array results

e.g. provide:
select GeneName,RecordID from mytsummeta where GeneID like '1NP017'
and receive an array of the values in the select field, so in this
case receive ATRX and the record ID
```

Source: `http://npd.hgu.mrc.ac.uk/user/services`*, visited April 2014.*

[1] `https://www.biocatalogue.org/services/2062`, visited April 2014.

Throughout Chapters 3 and 4 we developed a text-centric generalised submission sifting workflow into a higher-order dataflow model designed specifically to support the profiling and matching of structured data in a web application context. Unfortunately, the central role of structured data in our motivating use cases and application domain makes it difficult to take advantage of the practical benefits of the relational model enjoyed in many other domains. Of all the structured data representation and access schemes discussed in Chapter 2, the relational model is the least well suited to implementing the SubSift and JSONMatch frameworks we described; document-based (e.g. NoSQL) and graph-based (e.g. RDF) schemes are both better suited and, indeed, each of our proof of concept implementations used document-based schemes.

Setting aside inherent problems with representing structured data in a relational format, the relational algebra's set query language data access scheme offers appealing engineering advantages over the imperative domain-specific language of our JSONMatch framework. A query language specifies what is to be achieved without specifying all the details of how it is to be achieved whereas an imperative language leaves all these details to the developer. Modern relational database management systems take advantage of this separation of 'what' from 'how' by applying sophisticated query planners that transparently rewrite the query to optimise resources, reducing execution time and memory requirements. By contrast, implementing equivalent optimisations in an imperative language setting requires changes to be made to the original program and, moreover, in every program where a particular optimisation is required. The development community's desire to bring the advantages of query languages to document-based Big Data and NoSQL systems has led to the development of Apache Hive[2], an SQL-like interface to the popular Hadoop MapReduce framework, directly competing with the imperative language Apache Pig[3] interface [GNC + 09, ORS + 08].

In this chapter, we speculate that our own higher-order dataflow model may similarly benefit from the addition of a relational query language interface. To investigate this idea we focus on the data comparison aspects of our uses cases and explore whether matching may be expressed through queries in a relational algebra defined on basic terms in the higher-order logic. We begin by reformulating Codd's relational algebra into a form that is closer to set theoretic mathematical conventions than traditional treatments of the algebra in textbooks. To lift this formulation to our higher-order representation we substitute relational sets for basic terms as the data representation, similarly upgrading each of the relational operators. The formalism we introduce in the chapter is the complete relational algebra upgraded for terms in a higher-order logic. Our approach to implementing matching in the algebra is to relax the join operator to be an approximate join and use this as the basis for matching and merging structured data. We demonstrate experimentally that a family of kernels and distances defined on basic terms show encouraging results as the mechanism for implementing matching as an approximate join.

[2]Apache Hive – `http://hive.apache.org/`, visited April 2014.
[3]Apache Pig – `http://pig.apache.org/`, visited April 2014.

5.1 Relational Model

Surveying the relational database literature can initially be confusing because authors adopt one of two alternative definitions of the relational representation. Indeed Codd himself switched from one representation to the other early on in his development of the relational model. The competing representations differ in their definition of the tuple: the earlier representation follows the conventional mathematical definition of a tuple whereas the later representation defines the, so called, *unordered labelled tuple*. In this chapter we follow Codd's original relational representation for reasons that we discuss below.

An n-tuple $(x_1,\ldots,x_n)$ is an element in the Cartesian product $D_1 \times \cdots \times D_n$ where $x_1,\ldots,x_n$ are values drawn from domains $D_1,\ldots,D_n$ respectively, $n \in \mathbb{N}$ and $n \geqslant 1$. We refer to n-tuples as *tuples* unless the value of n is significant. The core of the relational representation is the *relation*, which is a homogeneous set of such tuples defined as follows.

Definition 5.1.1 (Relation) *A* relation R *of degree* n *is a finite set of* n*-tuples such that* $R \subseteq D_1 \times \cdots \times D_n$ *where* $D_1,\ldots,D_n$ *are domains.*

The *schema* (or *scheme*) of a relation $R \subseteq D_1 \times \cdots \times D_n$ is denoted $R(D_1,\ldots,D_n)$ and R is said to be a relation on that schema. The domains $D_1,\ldots,D_n$ in a relation need not necessarily be distinct. All domain values are considered to be atomic, meaning that they are indivisible as far as the relational model is concerned. Also a special value called `NULL` is included in every domain[4].

Being a set, duplicate tuples are not permitted in a relation. This is a notable difference between the relational model and typical implementations of relational databases, where duplicate tuples are permitted in relations because a multiset (or bag) representation is used instead of a set representation[5]. In the relational database literature a relation is often referred to as a table, and a tuple as a record or row in a table. By definition, the rows of a table occur in a specific order whereas the elements of a set are entirely unordered and so a table is a rather inaccurate representation of a relation.

Prior to defining the relational operations and algebra it is first necessary to define some means for identifying values at specific positions within a tuple. Item $i \in \{1,\ldots,n\}$ of a tuple $t = (x_1,\ldots,x_n)$ is the value x_i and is referred to as $t|_i$. The set of tuple item indices for a given relation is the *relation index* of the relation.

Definition 5.1.2 (Relation Index) *The* relation index I_R *of a relation* R *of degree* n *is the set* $\{1,\ldots,n\}$.

The relation index referred to here should not be confused with the sorts of indexes used to increase record access speed within relational database implementations.

[4]The multiple roles of `NULL` values in the relational model fall outside the scope of this Thesis but a comprehensive discussion appears in [EN06].

[5]Relational databases typically follow SQL standards and consequently deviate from the relational model by allowing duplicate elements in relations, thereby invalidating the model's theoretical results [Cod90].

The relation index I_R for some relation R is isomorphic with the set of attribute names $\{\mathcal{A}_1,\ldots,\mathcal{A}_n\}$ in the traditional *name-based schema* $\mathcal{R}(\mathcal{A}_1,\ldots,\mathcal{A}_n)$, where $\mathcal{R}$ is the relation name associated with R, and $\mathcal{A}_1,\ldots,\mathcal{A}_n$ are attribute names associated with domains $D_1,\ldots,D_n$ respectively. For example, if `author(firstname, lastname, email)` is the name-based schema for a relation R, representing authors in a bibliographic database, then the set of attribute names $\{$`firstname`, `lastname`, `email`$\}$ is trivially isomorphic with the relation index $I_R = \{1,2,3\}$. Hence, without loss of generality, we use the relation index instead of the name-based schema in order to simplify the upgrading of the relational model to higher-order logic.

In keeping with our choice of the relation index, we also adopt Codd's original definition of the tuple for the elements of the relation as opposed to his subsequent alternative of the unordered labelled tuple. The latter relaxes the definition of a tuple so that the order of its items becomes irrelevant. More formally, an unordered labelled tuple u is defined as a set of name-value pairs such that $u = \{(\mathcal{A}_1,x_1),\ldots,(\mathcal{A}_n,x_n)\}$, where each $\mathcal{A}_i$ and x_i pair is a corresponding attribute name and item value. Pairing each value with an identifying attribute name brings the convenience of set operations on tuple items in defining the relational operations. Associativity and commutativity of various relational operations is achieved as a consequence but at the cost of the complexity of maintaining the attribute name uniqueness constraint, $\forall i,j \in 1\ldots n, \mathcal{A}_i = \mathcal{A}_j \implies i = j$, which in turn gives rise to a requirement for a *rename* operation [Mai83]. In our higher-order setting, as we later explain, operation associativity and commutativity cease to be relevant and so we are able to avoid this unnecessary complexity through our choice of tuple.

5.1.1 Relational Operations

The relational algebra defines eight operations on relations. Five of these operations (union, difference, product, projection and restriction) are considered fundamental, or primitive, as they can not be derived from combinations of the other relational operations. The remaining three (intersection, divide and join) can each be derived by combining the fundamental operations. The following sections define all eight operations, beginning with the fundamental ones. Practical implementations of the relational algebra also include additional convenience operations, aggregate operations and update operations that are not strictly part of the algebra and which we do not consider here.

Fundamental Relational Operations

The relational algebra defines the following five operations.

- relational union ($\cup$)
- relational difference ($-$)
- relational product ($\times$)
- relational projection (π)
- relational restriction (σ)

The definitions of relational union and difference (and, later when we discuss the non-fundamental operations, intersection) accord with the usual definition of the corresponding operations from set theory but apply solely to relations that are *union compatible*. Relations A and B are union compatible if and only if $A, B \subseteq D_1 \times \cdots \times D_n$. As would be expected from the normal definition of a set, all duplicate tuples are removed from the result (co-domain) of any of the following operations.

Definition 5.1.3 (Relational Union) *The* relational union $A \cup B$ *of relations* $A, B \subseteq D_1 \times \cdots \times D_n$ *is the relation* $A \cup B = \{t \mid t \in A \vee t \in B\}$.

Definition 5.1.4 (Relational Difference) *The* relational difference $A - B$ *of relations* $A, B \subseteq D_1 \times \cdots \times D_n$ *is the relation* $A - B = \{t \mid t \in A \wedge t \notin B\}$.

The definition of the relational product operation differs from the normal Cartesian product in that the result is a set of single tuples rather than a set of pairs of tuples. The single tuple is formed by applying a tuple concatenation function to the pair of tuples that would result from a conventional Cartesian product. *Tuple concatenation* is a binary function, *conc*, on tuples such that for tuples $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_m)$, $conc(a,b) = (a_1, \ldots, a_n, b_1, \ldots, b_m)$. The tuples a and b need not be union compatible.

Definition 5.1.5 (Relational Product) *The relational product* $A \times B$ *of relations* A *and* B *is* $A \times B = \{conc(a,b) \mid a \in A, b \in B\}$.

One consequence of our definition of relations is that the relational product is neither associative nor commutative. There does however exist a bijection between the tuple items of the different concatenated tuples. So $A \times B \equiv B \times A$ up to isomorphism. However, we will address the lack of associativity and commutativity later in this chapter.

Definition 5.1.6 (Relational Projection) *Let* A *be a relation with corresponding relation index* I_A. *Let* ρ *be a list of relation index items drawn from* I_A *such that* $\rho = [i_1, \ldots, i_n]$. *The* relational projection π *on* ρ *of* A *is:* $\pi_\rho(A) = \{(a|_{i_1}, \ldots, a|_{i_n}) \mid a \in A\}$.

Relational projection corresponds quite closely to `SELECT DISTINCT` in SQL.

Definition 5.1.7 (θ -Restriction) *Let* θ *be a predicate* $\theta : D \times D \to \mathbb{B}$ *for some domain* D. *For a relation* A, *with corresponding relation index* I_A, *the* θ-restriction σ_φ *of* A *is defined as follows.*

1. *If* φ *has the form* $i\,\theta\,j$ *where,* $i, j \in I_A$ *and* $a|_i, a|_j \in D$, *then*

$$\sigma_{i\,\theta\,j}(A) = \{a \mid a \in A \ \wedge \ a|_i \,\theta\, a|_j \ \}.$$

2. *If* φ *has the form* $i\,\theta(v)$, *where* $i \in I_A$ *and* $a|_i, v \in D$, *then*

$$\sigma_{i\,\theta(v)}(A) = \{a \mid a \in A \ \wedge \ a|_i \,\theta\, v \ \}.$$

The infix θ in the subscript of σ follows the historical convention from the relation database literature and so $i\,\theta\,j$, or equivalently $\theta(i,j)$, does not mean that θ applies to i and j; instead $i\,\theta\,j$ is shorthand notation for the membership test $a|_i\,\theta\,a|_j$ for all $a \in A$. Also, the parenthesised subscript (v) is our notation to distinguish v as a literal value as distinct to the index of a tuple item, such as i, where no parentheses are used.

The predicate θ is typically drawn from the set $\{=, \neq, <, \leqslant, >, \geqslant\}$ but does not necessarily have to come from this set. θ-restriction is often just referred to as *restriction* and in such cases θ is assumed to be the equality operation. The name *selection* is often used instead of *restriction* in the literature but we will not use this synonym here to avoid confusion with the `select` operation from SQL which has a somewhat different meaning. In fact, restriction corresponds more closely to the `WHERE` clause in SQL.

Definition 5.1.8 (Generalised Restriction) *Let φ be a proposition that consists of atoms as allowed in θ-restriction and the logical operations $\wedge$, $\vee$ and $\neg$. If A is a relation then the* generalised restriction *σ_φ is defined on A as follows:*

$$\sigma_\varphi(A) = \{\, a \mid a \in A \;\wedge\; \varphi(a) \,\}.$$

Proposition 5.1.9 *Generalised restriction does not increase the expressive power of the relational algebra if the algebra already includes θ-restriction.*

Proof 5.1.10 *Let φ and ψ be propositions on elements of relation A and let them consist only of atoms as allowed in θ-restriction. If we assume the usual set intersection, union and difference operations then the result follows directly from the equivalences: $\sigma_{\varphi\wedge\psi}(A) = \sigma_\varphi(A) \cap \sigma_\psi(A)$, $\sigma_{\varphi\vee\psi}(A) = \sigma_\varphi(A) \cup \sigma_\psi(A)$ and $\sigma_{\neg\varphi}(A) = A - \sigma_\varphi(A)$.*

Derivative Relational Operations

The following three derivative relational operations can be expressed solely as combinations of the five fundamental relational operations.

- relational intersection ($\cap$)
- relational division ($\div$)
- θ-join ($\bowtie$)

Two further derivative operations, generalised join ($\bowtie_\varphi$) and natural join ($\bowtie_{i,j}$), are commonly used convenience forms of the more general θ-join. In the remainder of this section we define and discuss all five of these non-fundamental operations.

Definition 5.1.11 (Relational Intersection) *The* relational intersection *$A \cap B$ of relations $A, B \subseteq D_1 \times \cdots \times D_n$ is the relation $A \cap B = \{t \mid a \in A \wedge t \in B\}$.*

Expressed in terms of the fundamental relational operations $A \cap B = A - (A - B)$, $A \cap B = B - (B - A)$ and $A \cap B = A \cup B - (A - B) - (B - A)$.

Relational division is the reverse of the relational product.

Definition 5.1.12 (Relational Division) *The* relational division $A \div B$ *of relation* $A \subseteq D_1 \times \cdots \times D_n \times D_{n+1} \times \cdots \times D_m$ *and relation* $B \subseteq D_{n+1} \times \ldots D_m$, *where* $n \leqslant m$, *is the* n*-ary relation*

$$A \div B = \{\, t \mid \forall b \in B, conc(t,b) \in A \}.$$

Expressed in terms of the fundamental relational operations,

$$A \div B = \pi_{1,\ldots,n}(A) - \pi_{1,\ldots,n}((\pi_{1,\ldots,n}(A) \times B) - A).$$

Definition 5.1.13 (θ -Join) *Let* θ *be a predicate* $\theta : D \times D \to \mathbb{B}$ *for some domain* D. *If* A *and* B *are relations with tuple items* $a|_i \in D$ *and* $b|_j \in D$ *respectively for some* $(i,j) \in I_A \times I_B$, *then the* θ -join $\bowtie_{i\,\theta\,j}$ *of* A *and* B *is defined as*

$$A \bowtie_{i\,\theta\,j} B = \sigma_{i\,\theta\,j}(A \times B).$$

When θ is equality the θ -join is called the *equi-join*. By replacing the θ -restriction operation in the θ -join by the generalised restriction operation we arrive at the definition of the *generalised join*.

Definition 5.1.14 (Generalised Join) *Let* φ *be a proposition that consists of atoms as allowed in* θ *-restriction and the logical operations* $\wedge$, $\vee$ *and* $\neg$. *If* A *and* B *are relations then the* generalised join $\bowtie_\varphi$ *is defined as*

$$A \bowtie_\varphi B = \sigma_\varphi(A \times B).$$

For the purposes of upgrading relational joins to handle structured data, it is sufficient to consider just the θ -join and, optionally as a useful syntactic convenience, the generalised join. However, a description of relational joins would not be complete without mentioning the, so called, *natural join*. The natural join is the result of a projection of an equi-join such that duplicate tuple items are removed from the resulting relation.

Definition 5.1.15 (Natural Join) *Let* A *and* B *be relations with tuples* $a \in A$ *and* $b \in B$. *If tuple items* $a|_i \in D$ *and* $b|_j \in D$ *for some domain* D *and some* $(i,j) \in I_A \times I_B$ *then the* natural join $\bowtie_{(i,j)}$ *of* A *and* B *is*

$$A \bowtie_{(i,j)} B = \pi_\rho(\sigma_{i=j}(A \times B)),$$

where $\rho = I_A \cup \{\, |I_A| + \ell \mid \ell \in I_B \wedge \ell \neq j \,\}$.

When unqualified reference is made in the literature to a "*relational join*", this typically refers to the *natural join*. The literature also defines several other types of join, including *semijoin*, *antijoin*, *outer joins* and *inner joins*. These may all be expressed in terms of the fundamental operations described in this chapter but are not discussed further here and the interested reader is instead referred to standard texts such as [Cod90, Dat91]. For the purposes of upgrading relational joins to handle structured data, it is sufficient to consider just the θ -join and optionally, as a useful syntactic convenience, the generalised join.

Natural joins as traditionally defined in Codd's relational algebra [Cod70, Cod79] are both commutative, i.e. $A \bowtie B = B \bowtie A$, and associative, i.e. $(A \bowtie B) \bowtie C = A \bowtie (B \bowtie C)$. Neither of these properties holds for the relational joins described in this chapter. In the context of traditional database applications, both of these properties are of practical importance. By contrast, in the context of joining structured data rather than relational data, the resultant structured data would not typically be expected to have the same structure as the original data and thus commutativity and associativity become less relevant. Also, our restatement of relational joins is closer to the usual mathematical notions of a relation and is closer to Codd's earliest published definition [Cod69].

5.1.2 Relational Algebra

Given the relational representation and fundamental operations defined above we can now define the *relational algebra* along similar lines to [Mai83] but with alterations to accommodate our choice of representation.

Definition 5.1.16 (Relational Algebra) *Let $\mathcal{D}$ be a set of domains. Let dom be a total function from values $v \in D$ to their associated domain $D \in \mathcal{D}$ such that $dom(v) = D$. Let R be a set of relations such that $R = \{R \mid R \subseteq D_1 \times \cdots \times D_n \wedge D_i \in \mathcal{D},\ i = 1..n\}$. Let ind be a total function from relations $R \in R$ to relation indexes $I_R \subseteq \mathbb{N}^+$. Let Θ be a set of predicates over domains in $\mathcal{D}$, and the logical operations $\wedge$, $\vee$ and $\neg$ over the booleans. Let O be the set of operations: relational union $\cup$, relational difference $-$, relational product $\times$, relational projection π, and relational restriction σ. The* relational algebra *over $\mathcal{D}$, dom, R, ind, Θ and O is the 6-tuple $\mathcal{R} = (\mathcal{D}, dom, R, ind, \Theta, O)$. An* algebraic expression *over $\mathcal{R}$ is any expression formed legally, according to the definitions of the operations, from the relations in R using the operations in O.*

The derivative operations, such as relational intersection ($\cap$), θ-join ($\bowtie_\theta$), relational division ($\div$) and the natural join ($\bowtie$), are not defined as members of O because they may be defined from combinations of the fundamental operations.

Index set I_R of each relation $R \in R$ does not appear in the definition directly, as would be the case with name-based relational schema, but is instead obtained through the function *ind* so that an association between relation and index is maintained.

The function *dom* ensures that every value in every domain in $\mathcal{D}$ is associated with its domain. This association between a value and its domain is an important feature in the upgrading of relational algebra to the higher-order logic where we exploit a similar relationship between a value and its type (to be defined).

5.2 Lifting the Relational Model to a Higher-Order

The relational model is the *de facto* standard for database-driven applications, including numerous existing web applications, web services and workflow systems. Despite this, the relational model is not ideally suited for representing semi-structured data such as Web pages, XML, JSON and numerous other document-based formats.

Neither is the relational model convenient for representing structured data such as trees, lists, bags and so on, although the representation of such structures in relational databases is commonplace using a multitude of (often tortuous) representations and querying patterns, some of which we described in "2.2.1 Relational Representations" [Dat91, EN06, GMN84, Mai83]. To overcome these difficulties with the relational representation, we adopt the same *individuals-as-terms* knowledge representation employed in "Chapter 4 – Higher-Order Dataflows".

Recall that the individuals-as-terms representation is a generalisation of the relational model's attribute-value representation and collects all information about an individual in a single term. As such, the individuals-as-terms representation has much in common with the various hierarchical and object database models that were the forerunners of the highly successful relational model that all but replaced them – until the recent Big Data driven emergence of NoSQL databases caused a resurgence of interest in non-relational data models. The terms in question are the *basic terms* from a family of typed terms in the higher-order logic based on Church's simple theory of types with several extensions [Llo02] and summarised in Chapter 2. In contrast to the relational model, the individuals-as-terms model offers straight forward representations of both structured and semi-structured data while at the same time having the representational capacity to represent relations from the relational model. This representational flexibility makes the individuals-as-terms model a convenient representation of heterogeneous data, enabling the collection of all information about an individual in a single term irrespective of whether that information is relational, semi-structured or structured.

When working with terms and basic terms it is useful or, as in the context of the relational algebra on basic terms, necessary to be able to refer to sub-parts of a term either individually or collectively. We refer to the process of specifying a specific sub-part or set of sub-parts as *indexing*.

5.2.1 Indexing Basic Terms

In the logic, sub-parts of a term are referred to as subterms and so we are concerned with indexing the subterms of a basic term. The standard method for indexing subterms in the logic enumerates a decomposition of a given term such that every subterm is labelled with a unique string [Llo03]. However, we introduce an alternative approach to indexing that, instead of enumerating all subterms of a term, defines a *type tree index set* over all subtypes of the type of a basic term. To do this we first adopt the definition of a type tree from [GF05] and then define a different annotation of the tree such that every member of the type tree index set identifies a set of terms rather than a single term. This ensures any index defined on a type is meaningful across all terms of that type. Furthermore, the set of subterms identified is guaranteed to consist entirely of well-formed basic terms.

Below we consider two subterm indexing methods which we call *term-based indexing* and *type-based indexing*, the latter of which has a variant which we also discuss.

Term-based Indexing

We refer to *term-based indexing* as the approach taken in [Llo03], whereby the *occurrence set* $\mathscr{O}(t)$ of a term t is defined to enable subterm indexing such that the subterm of t at occurrence $o \in \mathscr{O}(t)$ may be referenced as $t|_o$. Each occurrence is either a numeric string that uniquely labels a subterm of a given term, or is the empty string ε, labelling the reflexive subterm. Example 5.2.1, using the trivial case of a basic tuple, informally illustrates subterm indexing based on a term's occurrence set as defined in [Llo03].

Example 5.2.1 *If basic term t is the tuple $t = (A,B,C,D)$ such that $A,B,C,D \in \mathfrak{B}$, then the occurrence set of t is $\mathscr{O}(t) = \{\varepsilon,1,2,3,4\}$, and the subterms of t are indexed as $t|_\varepsilon = (A,B,C,D)$, $t|_1 = A$, $t|_2 = B$, $t|_3 = C$ and $t|_4 = D$.*

For the case of basic tuples, such a scheme is a suitable analogue of the relation index, because indexing over the items of basic tuples corresponds closely with numeric indexing over the items of tuples in the traditional relation index. If, however, we move beyond indexing basic tuples and onto basic structures or basic abstractions, then indexing directly on the terms themselves has two undesirable consequences, in our context, which we outline below.

Firstly, indexes over basic structures and basic abstractions requires foreknowledge of the extension of data instances in order to select a meaningful occurrence that applies to all the basic terms in a basic term relation, as illustrated for basic structures in Example 5.2.2. In other words, some occurrence values are local to a subset of the basic terms in the relation: possibly even local to a single term.

Example 5.2.2 *If basic terms $s,t \in \mathfrak{B}_{List\,M}$ are the lists $s = [A,B,C]$ and $t = [A,D]$, where $A,B,C,D : M$, and M is a nullary type constructor, then the occurrence sets of s and t are $\mathscr{O}(s) = \{\varepsilon,1,2,21,22,221,222\}$ and $\mathscr{O}(t) = \{\varepsilon,1,2,21,22\}$, the derivation of which can be seen from Figure 5.1. The occurrence sets correspond to the subterms $s|_\varepsilon = [A,B,C]$, $s|_1 = A$, $s|_2 = [B,C]$, $s|_{21} = B$, $s|_{22} = [C]$, $s|_{221} = C$, $s|_{222} = []$, and $t|_\varepsilon = [A,D]$, $t|_1 = A$, $t|_2 = [D]$, $t|_{21} = D$, $t|_{22} = []$. So, for instance, occurrence 21 can be used to index both s and t because $21 \in \mathscr{O}(s) \cap \mathscr{O}(t)$. In contrast, occurrence 221 can only be used to index s because $221 \in \mathscr{O}(s)$, but $221 \notin (\mathscr{O}(s) \cap \mathscr{O}(t))$. Thus occurrence 221 is local to s with respect to t.*

This *indexing locality problem* increases as the complexity and nesting of basic terms increases. Notably for abstractions, representing sets and multisets etc., the locality of occurrences is total; an occurrence has little or no meaning as an index outside of the abstraction term itself, as can be seen from Example 5.2.3.

Example 5.2.3 *Let basic terms $s,t \in \mathfrak{B}_{\alpha\to\Omega}$ be the sets $s = \{A,B,C\}$ and $t = \{B,D\}$, represented by the basic abstractions*

$s = \lambda x.\ \mathit{if}\ x = A\ \mathit{then}\ \top\ \mathit{else\ if}\ x = B\ \mathit{then}\ \top\ \mathit{else\ if}\ x = C\ \mathit{then}\ \top\ \mathit{else}\ \bot$,
$t = \lambda x.\ \mathit{if}\ x = B\ \mathit{then}\ \top\ \mathit{else\ if}\ x = D\ \mathit{then}\ \top\ \mathit{else}\ \bot$,

where $A,B,C,D : M$, and M is a nullary type constructor. Given that " if p then q

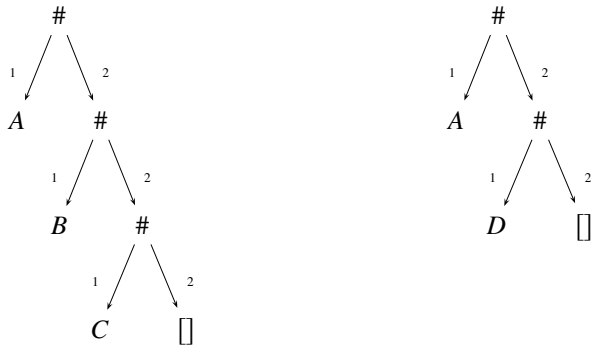


Figure 5.1: Term-based indexing for basic structures representing lists $[A,B,C]$ and $[A,D]$, which is notational sugar for $A\#B\#C\#[]$ and $A\#D\#[]$, where # and [] are the usual list data constructors, $A,B,C,D:M$, and M is a nullary type constructor.

else r " is infix notation for $if\_then\_else(p,q,r)$*, these basic abstractions may also be rewritten as follows, abbreviating " if_then_else " to " ite " for conciseness:*

$s = \lambda(x, ite(=(x,A), \top, ite(=(x,B), \top, ite(=(x,C), \top, \bot))))$,
$t = \lambda(x, ite(=(x,B), \top, ite(=(x,D), \top, \bot)))$.

Their occurrence sets are thus $\mathscr{O}(s) = \{\varepsilon, 1, 2, 21, 22, 23, 211, 212, 231, 232, 233, 2331, 2332, 2333, 23311, 23312\}$ *and* $\mathscr{O}(t) = \{\varepsilon, 1, 2, 21, 22, 23, 211, 212, 231, 232, 233, 2311, 2312\}$, *the derivation of which can be seen from Figure 5.2. Out of these occurrences, the only ones likely to be of practical use in a join operation correspond to the subterms* $s|_{212} = A$, $s|_{2312} = B$, $s|_{23312} = C$ *and* $t|_{212} = B$, $t|_{2312} = D$. *These occurrences are clearly local to the term upon which they are defined.*

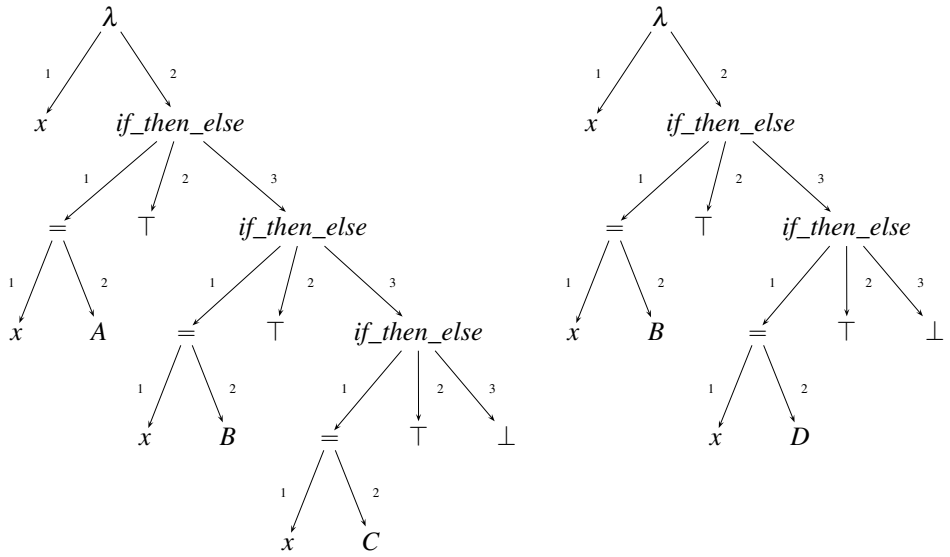


Figure 5.2: Term-based indexing for basic abstractions representing sets $\{A,B,C\}$ and $\{B,D\}$, where $A,B,C,D:M$, and M is a nullary type constructor.

The second undesirable consequence, in our context, of indexing directly on the terms using the occurrence set defined in [Llo03] is that subterms are not guaranteed to be basic terms. For example, in any basic abstraction,

$$\lambda x.\ \text{if } x = t_1 \text{ then } s_1 \text{ else } \ldots \text{ if } x = t_n \text{ then } s_n \text{ else } s_0 \in \mathfrak{L},$$

the variable x is a subterm but is not a basic term. This is technically the case by the definitions of basic terms and subterms, Definition 2.3.7 (p.53) and Definition 2.3.13 (p.56) respectively, but also intuitively true as x is not a ground term, unlike all the other terms $t_1, \ldots, t_n$ and $s_1, \ldots, s_n, s_0$ in the above expression.

In the context of exact relational joins, including terms that are not basic terms is not of itself problematic because the logic has a well defined equality operation over all terms, and all subterms are terms. The equality operation in the logic is defined inductively such that it recursively compares subterms of a basic term and all its constituent subterms. Thus, continuing our example of x in basic abstractions, every basic abstraction begins with λx and so these subterms will always be equal when comparing any pair of basic abstractions, irrespective of the rest of the subterms. This part of the equality comparison is entirely redundant and it is only in the rest of the subterms in a pair of basic abstractions that will determine equality or inequality. Although not necessarily of practical importance, the semantics of comparing two instances of x in different terms is not entirely clear and the same would be true of any join based on this equality operator.

However, leaving aside issues of redundancy or the semantics of joining on such subterms, for our approximate joins, admitting subterms that are not basic terms prevents the application of existing approximate equality operations defined only for the subset of terms that are basic terms. In other words, the theory supporting the use of kernels defined on basic terms does not cover other types of term, such as x in our example above. We suspect that it may be technically possible to extend the kernel for basic terms to cover all terms but semantically there seems little value in doing this (for our work) because the strength of the basic terms approach is its representation of individuals as strongly-typed ground terms in an intuitive way that enables kernels to be designed by associating them with types. Extending the theory into more abstract semantics that are, arguably, just artefacts of the logic would depart from this approach and reduce its intuitive strengths.

Type-based Indexing

To solve both of the aforementioned problems of term-based indexing, we introduce an alternative approach to indexing that, instead of defining an occurrence set over all subterms of a term, defines a *type tree index set* over all subtypes of the type of a basic term. To do this we first adopt the definition of a type tree from [GF05] and then define a specific annotation of the tree such that every member of the type tree index set identifies a set of terms rather than a single term. The set of terms identified is guaranteed to consist entirely of basic terms. Moreover, this type-based indexing overcomes the indexing locality problem of term-based indexing.

To achieve this we follow the same interpretation of subtypes as [Llo03] and restrict our attention to basic terms whose basic structures are in canonical form as

defined below.

Definition 5.2.4 (Basic Structures in Canonical Form) *A type* $\tau = T\ \alpha_1 \ldots \alpha_k$ *is a basic structure in canonical form when, for all data constructors* $C_i : \tau_{i1} \rightarrow \cdots \rightarrow \tau_{in} \rightarrow \tau$ *that are associated with* T*, all the types of* τ_{ij} *are subtypes of* τ*.*

We begin our definition of the type tree index set with some preparatory notation. Let $\mathbb{Z}^+$ denote the set of positive integers and $(\mathbb{Z}^+)^*$ the set of all strings over the alphabet of positive integers, with ε denoting the empty string. io denotes the string concatenation of i with o where $i \in \mathbb{Z}^+$ and $o \in (\mathbb{Z}^+)^*$.

Definition 5.2.5 (Type Tree Index Set) *The* type tree index set *of a canonical type* τ*, denoted* $\mathscr{O}(\tau)$*, is the set of strings in* $(\mathbb{Z}^+)^*$ *defined inductively on the structure of* τ*.*

1. *If* τ *is an atomic type, then* $\mathscr{O}(\tau) = \{\varepsilon\}$*.*
2. *If* τ *is a basic structure type* $\tau = T\ \alpha_1 \ldots \alpha_n$ *in canonical form, with data constructors* $C_i : \tau_{i_1} \rightarrow \cdots \rightarrow \tau_{i_m} \rightarrow \tau$ *for all* $i \in \{1, \ldots, l\}$*, then* $\mathscr{O}(\tau) = \{\varepsilon\} \cup \bigcup_{v=1}^{p} \{vo_v \mid o_v \in \mathscr{O}(\xi_v)\}$*, where* $\xi_1, \ldots, \xi_p$ *are the types from* α_k *where* $\alpha_k = \tau_{i_j}$ *and* $\tau_{i_j} \neq \tau$*, and assuming that for every* $\tau_{i_j} \neq \tau$ *there exists an* α_k *such that* $\alpha_k = \tau_{i_j}$*.*
3. *If* τ *is a basic abstraction type* $\beta \rightarrow \gamma$*, then* $\mathscr{O}(\tau) = \{\varepsilon\} \cup \{1o \mid o \in \mathscr{O}(\beta)\} \cup \{2o \mid o \in \mathscr{O}(\gamma)\}$*.*
4. *If* τ *is a basic tuple type* $\tau = \tau_1 \times \cdots \times \tau_n$*, then* $\mathscr{O}(\tau) = \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{io_i \mid o_i \in \mathscr{O}(\tau_i)\}$*.*

Part 1, the base case, states that types for which all the associated data structures have arity zero, such as Ω (the type of the booleans), *Int* (the type of the integers), and *Char* (the type of characters), have a singleton index set containing the empty string. Part 2 states that each subtype that occurs in the signatures of the associated data constructors, and that is not itself of type τ of the basic structure, is labelled with a unique string. Part 3 labels the β and γ types of basic abstractions with a pair of unique strings. Similarly, part 4 labels each tuple item in a basic tuple with a unique string.

Definition 5.2.6 (Subtype at a given type tree index) *The subtype of a canonical type* τ *at type tree index* $o \in \mathscr{O}(\tau)$*, denoted* $\tau]_o$*, is the type that occurs in* τ *at* o*.*

The significance of defining indexing on the type tree of basic terms rather than on the terms themselves is that each member of a type tree index set $o \in \mathscr{O}(\tau)$ is not uniquely tied to any individual term of type τ. This increases the generality of the indexing such that each member of the type tree index set for type τ identifies, for any basic term $t : \tau$, an equivalence class of subterms rather than a single term. Thus $\mathscr{O}(\tau)$ induces a set of equivalence classes on the subterms of t. We refer to the set of subterms identified with a given member (index) of the type tree index set as the *basic subterm set* at that index.

Definition 5.2.7 (Basic Subterm Set) *If t is a basic term of type τ and $o \in \mathscr{O}(\tau)$ then the* basic subterm set of t at type tree index o, *denoted $t|_o$, is defined inductively on the length of o as follows.*

1. *If $o = \varepsilon$, then $t|_o = \{t\}$.*
2. *If $o = jo'$, for some o', and t has the form $C\, t_1 \ldots t_m$, with associated type $T\, \alpha_1 \ldots \alpha_n$, then $t|_o = s_j|_{o'}$ where $s_j = t_i : \tau_i$ such that $\tau_i \neq \tau$ and $\tau_i = \alpha_j$.*
3. *If $o = 1o'$, for some o', and t has the form* if_then_else(u,v,s), *then $t|_o = u|_{o'} \cup s|_o$.*
4. *If $o = 2o'$, for some o', and t has the form* if_then_else(u,v,s), *then $t|_o = v|_{o'} \cup s|_o$.*
5. *If $o = io'$, for some o', and t has the form $(t_1, \ldots, t_n)$, then $t|_o = t_i|_{o'}$, for $i = 1, \ldots, n$.*

A basic subterm set *is a set of basic subterms of a basic term at some type tree index. A basic subterm is* proper *if it is not at type tree index ε.*

Basic subterms indexed in part 1, the base case, are singleton sets containing an atomic term. Basic subterms indexed in part 2 are basic structures. Basic subterms indexed in parts 3 and 4 are the support and value of basic abstractions, i.e. respective instances of α and β, from $\alpha \to \beta$. Basic subterms indexed in part 5 are basic tuples.

Below we give examples of a type tree index set and basic subterm sets for each of basic tuples, basic structures, and basic abstractions. Starting with basic tuples in Example 5.2.8 where it can be seen that in comparison to the term-based indexing from Example 5.2.1, type-based indexing identifies all the same terms, but as singleton sets and in addition it identifies the reflexive term at $t|_\varepsilon$.

Example 5.2.8 *If basic tuple $t \in \mathfrak{B}_{M \times N \times O \times P}$ is the term $t = (A,B,C,D)$, where $A : M$, $B : N$, $C : O$, $D : P$, then the type tree index set of t is $\mathscr{O}(t) = \{\varepsilon, 1, 2, 3, 4\}$, the derivation of which can be seen from Figure 5.3. The basic subterm sets of t are $t|_\varepsilon = \{(A,B,C,D)\}$, $t|_1 = \{A\}$, $t|_2 = \{B\}$, $t|_3 = \{C\}$ and $t|_4 = \{D\}$.*

Representing basic structures, the usual right branching representation of lists is given in Example 5.2.9, where the basic subterm set at $t|_1$ captures one meaning of a list as a set of values and $t|_\varepsilon$ captures the meaning of a list as a set of sequences.

Example 5.2.9 *If τ is a type of lists such that $\tau = List\, M$, where $M \subseteq \mathfrak{B}$ is a nullary type constructor, with associated data constructors # and [], having signatures $[] : List\, M$, and $\# : M \to List\, M \to List\, M$, then the type tree index set of τ is $\mathscr{O}(\tau) = \{\varepsilon, 1\}$. If basic terms $s,t \in \mathfrak{B}_{List\, M}$ are the lists $s = [A,B,C]$ and $t = [A,D]$, then as can be seen from Figure 5.4, the basic subterm sets of s and t are $s|_\varepsilon = \{[A,B,C],[B,C],[C],[]\}$, $s|_1 = \{A,B,C\}$, and $t|_\varepsilon = \{[A,D],[D],[]\}$, $t|_1 = \{A,D\}$.*

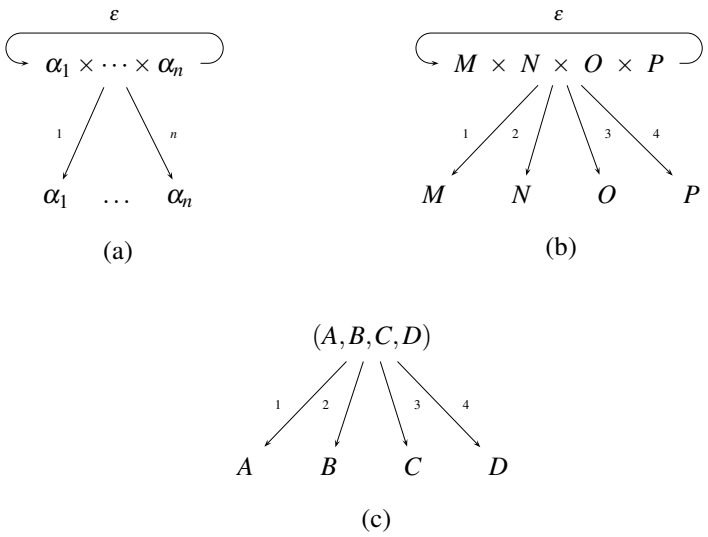


Figure 5.3: Type-based indexing for basic tuples. (a) Type tree index for n-tuples of type $\alpha_1 \times \cdots \times \alpha_n$. (b) Type tree index for 4-tuples of type $M \times N \times O \times P$. (c) Basic subterm tree for term (A,B,C,D) where $A : M$, $B : N$, $C : O$, $D : P$.

For basic abstractions, a set is given in Example 5.2.10 and a multiset in Example 5.2.11. For both sets and multisets, $t|_1$ captures the meaning as a set of values whereas $t|_2$ will always be $\{\top\}$ for sets and a set of multiplicities for multisets. A corollary of Definition 5.2.5 is that the type tree index set of a basic abstraction type is always $\{\varepsilon, 1, 2\}$.

Example 5.2.10 *If τ is a basic abstraction type representing sets such that $\tau = M \to \Omega$, where $M \subseteq \mathfrak{B}$ is a nullary type constructor, then the type tree index set of τ is $\mathscr{O}(\tau) = \{\varepsilon, 1, 2\}$. If basic term $t = \{A,B,C\}$, where $A,B,C : M$, then the basic subterm sets are $t|_\varepsilon = \{\{A,B,C\}\}$, $t|_1 = \{A,B,C\}$ and $t|_2 = \{\top\}$.*

Example 5.2.11 *If τ is a basic abstraction type representing multisets such that $\tau = M \to Nat$, where $M \subseteq \mathfrak{B}$ is a nullary type constructor and Nat is the type of the natural numbers, then the type tree index set of τ is $\mathscr{O}(\tau) = \{\varepsilon, 1, 2\}$. If basic term $t = \langle A,A,A,B,C,C\rangle$, where $A,B,C : M$, then the basic subterm sets are $t|_\varepsilon = \{\langle A,A,A,B,C,C\rangle\}$, $t|_1 = \{A,B,C\}$ and $t|_2 = \{1,2,3\}$.*

Proposition 5.2.12 *If τ is a basic abstraction type such that $\tau = \alpha \to \beta$, where $\alpha, \beta \in \mathfrak{B}$, then the type tree index set of τ is $\mathscr{O}(\tau) = \{\varepsilon, 1, 2\}$.*

Proof 5.2.13 *The result is a corollary of part 3 of Definition 5.2.5.*

Type Name-based Indexing

A useful and straight forward reformulation of type-based indexing is *type name-based indexing* that, instead of enumerating the edges of the type tree, directly labels

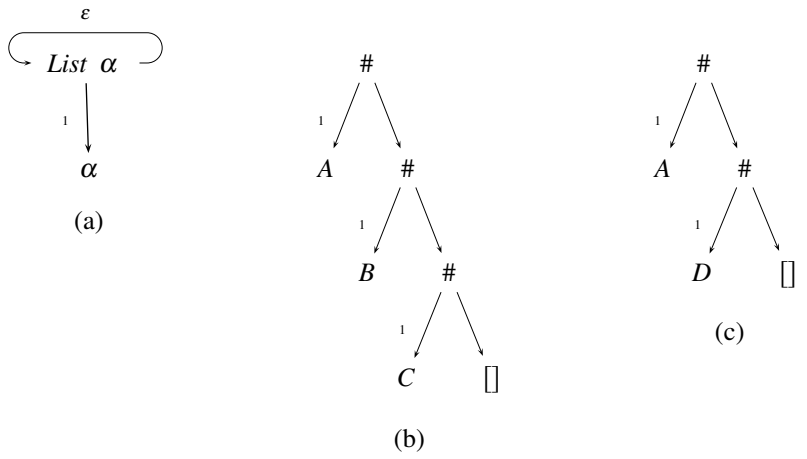


Figure 5.4: Type-based indexing for basic structures. (a) Type tree index for *List* α. (b) and (c) Basic subterm trees for terms $[A,B,C]$ and $[A,D]$ of type *List M* where $A,B,C,D : M$.

the vertices of the type tree. The simplest approach is to assign a unique type name to every vertex in the type tree. If the names assigned have no meaning to humans then this method offers no advantages over type-based indexing. However, if the knowledge representational formalism used to define types and data instances uses human-understandable names then type name-based indexing provides a useful notation for referring to basic subterm sets, as illustrated in Example 5.2.14.

Example 5.2.14 *Let Author be the type of authors from the publications domain, which define declaratively in the Haskell style syntax from [GLF04] as follows.*

```
type Author = (Name,Publications);
type Name = String;
type Publications = List Publication;
type Publication = (Mode,Coauthors,Title,Venue,Year);
data Mode = Journal | Proceedings | ... | Book;
type Coauthors = Coauthor -> Bool;
type Coauthor = String;
type Title = String;
type Venue = String;
type Year = Int;
```

This states that Author is a pair of Name and Publications, where Name is an alias for String the type of strings, and Publications is a list of publications, which in turn is a 5*-tuple of Mode, Coauthors, ..., Year, where Mode has the nullary data constructors Journal, Proceedings, ..., Book, and so on through to Year which is an alias for the type Int, the type of the integers. Coauthors is a basic abstraction from Coauthor to Bool, where Bool is the type* Ω*, i.e. Coauthors is a set of coauthors. To ensure the required uniqueness of type names Coauthor, Title, Venue and Name are aliases for the type String. The type tree index set is thus*

$$\{\, \mathit{Author},\ \mathit{Author.Name},\ \ldots,\ \mathit{Author.Publications.Publication.Year} \,\}.$$

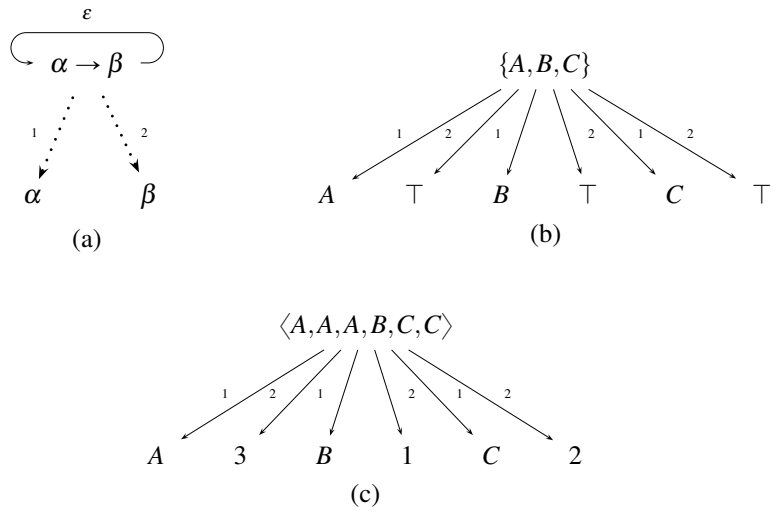


Figure 5.5: Type-based indexing for basic abstractions. (a) Type tree index for type $\alpha \to \beta$. (b) Basic subterm tree for set $\{A,B,C\}$, type $M \to \Omega$, where $A,B,C : M$ and $\top : \Omega$. (c) Basic subterm tree for multiset $\langle A,A,A,B,C,C\rangle$, type $M \to \text{Nat}$, where $A,B,C : M$ and $1,2,3 : \text{Nat}$.

A type tree index set generated using this method is isomorphic with that produced by Definition 5.2.5, as illustrated informally in Figure 5.6. The constraint that all basic subtypes must be uniquely named permits the following simpler definition of a basic subterm set.

Definition 5.2.15 (Basic Subterm Set (with named types)) *If t is a closed basic term of type τ and $\alpha \subseteq \tau$ then the* basic subterm set of t at type α*, denoted $t|_\alpha$, is $t|_\alpha = \{s \mid s \text{ occurs in } t \text{ with type } \alpha\}$. A* basic subterm set *is a set of basic subterms of a basic term at some type tree $\alpha \subseteq \mathfrak{B}$. A basic subterm is* proper *if $\alpha \neq \tau$.*

5.3 Basic Term Relational Model

We now upgrade the relational model for structured data. The way we achieve this is to first upgrade the knowledge representation of the relation to be a set of basic terms rather than the traditional set of tuples. We then upgrade the relation index so that it indexes parts of a basic term rather than the traditional parts of a tuple. Once these two steps are completed, upgrading the relational operations follows almost automatically with only modest changes to the definitions of the θ-restriction and joins. So to begin, we first upgrade the relation from section 5.1.1 to become the *basic term relation* [PF13c].

5.3.1 Basic Term Relational Representation

Our basic term relational representation is based on the *basic term relation*, which is a higher-order analogue of the traditional relation and is defined as follows.

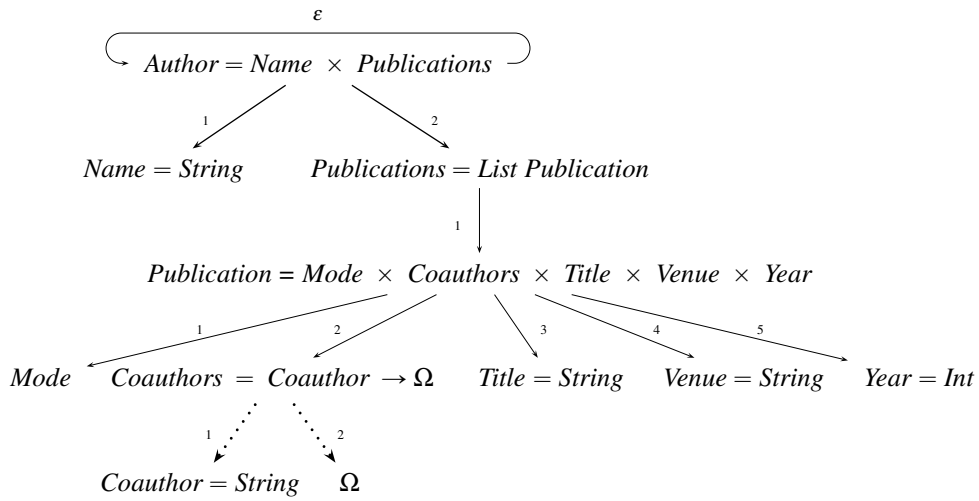


Figure 5.6: Type name-based and type-based indexing for type *Author*.

Definition 5.3.1 (Basic Term Relation) *A* basic term relation $R \subseteq \mathfrak{B}_\alpha$ *is a finite set of basic terms for some given type* $\alpha \in \mathfrak{S}^c$.

In the above definition of the basic term relation we describe R as a subset of $\mathfrak{B}_\alpha$ in order to emphasise the intended meaning of R as a set. Doing so also maintains a clear syntactic similarity to the definitions from the traditional relational algebra. However, in the higher-order logic we could equally have written $R \in \mathfrak{B}_{\alpha\to\Omega}$ because, being a set, R is a basic abstraction of type $\alpha \to \Omega$, where Ω is the type of the booleans. Thus a basic term relation is itself a basic term.

5.3.2 Indexing Basic Term Relations

Having upgraded our representation of a relation $R : \tau$ to handle structured data represented as basic terms, and having chosen a suitable indexing method for the basic subterm set $\mathscr{O}(\tau)$, we are now able to conveniently define the *basic term relation index* as the structured data counterpart of the relation index.

Definition 5.3.2 (Basic Term Relation Index) *The* basic term relation index I_R *of a basic term relation* R *of type* τ *is* $I_R = \mathscr{O}(\tau)$.

5.3.3 Basic Term Relational Operations

We present the fundamental basic term relational operations first before going on to define the derivative basic term relational operations.

Fundamental Basic Term Relational Operations

The basic term relational algebra defines five operations that directly correspond to the same five fundamental operations defined in relational algebra.

- basic term union ($\cup$)
- basic term difference ($-$)
- basic term product ($\times$)
- basic term projection (π)
- basic term restriction (σ)

Each of these operations on basic term relations is defined and discussed throughout the remainder of this section. The definitions of basic term union and difference (and, later when we discuss the non-fundamental operations, intersection) accord the usual set theory but apply solely to sets that are basic term relations and are relations of the same type.

Definition 5.3.3 (Basic Term Union) *The* basic term union $A \cup B$ *of basic term relations* $A \subseteq \mathfrak{B}_\alpha$ *and* $B \subseteq \mathfrak{B}_\alpha$, *for some* $\alpha \in \mathfrak{S}^c$, *is the basic term relation* $A \cup B = \{t \mid t \in A \vee t \in B\}$.

Definition 5.3.4 (Basic Term Difference) *The* basic term difference $A - B$ *of basic term relations* $A \subseteq \mathfrak{B}_\alpha$ *and* $B \subseteq \mathfrak{B}_\alpha$, *for some* $\alpha \in \mathfrak{S}^c$, *is the basic term relation* $A - B = \{t \mid t \in A \wedge t \notin B\}$.

The basic term product is, potentially, more troublesome to define than its corresponding operation from traditional relational algebra in that there is no single natural structure for combining arbitrary basic terms that is the analogue of concatenating a pair of relational tuples into a single relational tuple. As no one structure seems more appropriate than another here, we simply adopt the traditional mathematical Cartesian product as this is both sufficient and straight forward.

Definition 5.3.5 (Basic Term Product) *The basic term product* $A \times B$ *of basic term relations* $A \subseteq \mathfrak{B}$ *and* $B \subseteq \mathfrak{B}$ *is their Cartesian product such that* $A \times B = \{(a, b) \mid a \in A,\ b \in B\}$.

Note that duplicates are removed from the basic term relational product, as is normal for sets, and so $|A \times B| \leqslant |A||B|$. Also, there is no requirement for A and B to be of the same type; if the type of A is α and of B is β then the type of $A \times B$ is $\alpha \times \beta$.

Definition 5.3.6 (Basic Term Projection) *If* $t \in \mathfrak{B}$ *then the* basic term projection π *of* t *on* $i \in I_t$ *is*

$$\pi_i(t) = \{s \mid s \text{ is the basic subterm of } t \text{ at type tree index } i\}.$$

A basic term projection $\pi_i(t)$ may also be written as $t|_i$.

Basic term projection is defined over basic terms rather than just a basic term relation and so is more general than relational projection from traditional relational algebra. Thus, the same basic term projection operation may be applied to both an entire basic term relation or to an individual member of a basic term relation.

Definition 5.3.7 (Basic Term θ -Restriction) *Let θ be a predicate $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ for some $\alpha \in \mathfrak{S}^c$. If A and B are basic term relations with basic terms $a|_i \subseteq \mathfrak{B}_\alpha$ and $b|_j \subseteq \mathfrak{B}_\alpha$ respectively for some $(i,j) \in I_A \times I_B$, then* basic term θ -restriction $\sigma_{i\theta j}$ *is defined on $T \subseteq A \times B$ as*

$$\sigma_{i\theta j}(T) = \{(a,b) \mid a|_i \,\theta\, b|_j \ \wedge \ (a,b) \in T\}.$$

The predicate $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ is defined on sets of basic terms. In other words, θ is a binary predicate on basic term relations. The basic term restriction, like its counterpart in traditional relational algebra, produces relations of the same type as the type to which it is applied and of cardinality such that $|\sigma_{i\theta j}(T)| \leqslant |T|$.

Definition 5.3.8 (Basic Term Generalised Restriction) *Let φ be a proposition that consists of atoms as allowed in basic term θ -restriction and the logical operations $\wedge$, $\vee$ and $\neg$. If A and B are basic term relations then the* basic term generalised restriction σ_φ *is defined on $T \subseteq A \times B$ as*

$$\sigma_\varphi(T) = \{\, t \mid \varphi(t) \ \wedge \ t \in T\}.$$

Having defined the fundamental basic term relational operations we now turn to the derivative basic term relational operations.

Derivative Basic Term Relational Operations

The remaining three operations on basic term relations are also counterparts of their corresponding operations in traditional relational algebra and may be defined solely in terms of the five fundamental basic term relational operations.

- basic term intersection ($\cap$)
- basic term division ($\div$)
- basic term join ($\bowtie$)

In the remainder of this section we define and discuss each of these non-fundamental operations and explain why they are conceptually useful in their own right.

Definition 5.3.9 (Basic Term Intersection) *The* basic term intersection $A \cap B$ *of basic term relations $A \subseteq \mathfrak{B}_\alpha$ and $B \subseteq \mathfrak{B}_\alpha$, for some $\alpha \in \mathfrak{S}^c$, is the basic term relation $A \cap B = \{t \mid t \in A \wedge t \in B\}$.*

Expressed in terms of the fundamental relational operations $A \cap B = A - (A - B)$, $A \cap B = B - (B - A)$ and $A \cap B = A \cup B - (A - B) - (B - A)$.

Basic term division is the reverse of basic term product[6].

[6]We denote division as ' $\div$ ' rather than ' / ' to avoid confusion with the latter's established prior use in *parameter/type* binding notation from type substitutions in the higher-order logic.

Definition 5.3.10 (Basic Term Division) *The* basic term division $A \div B$ *of basic term relations* $A \subseteq \mathfrak{B}_\alpha \times \mathfrak{B}_\beta$ *and* $B \subseteq \mathfrak{B}_\beta$, *for some* $\alpha \in \mathfrak{S}^c$, *is the basic term relation*

$$A \div B = \{a \mid \forall b \in B, (a,b) \in A\}.$$

Expressed in terms of the fundamental relational operations,

$$A \div B = \pi_1(A) - \pi_1((\pi_1(A) \times B) - A).$$

Recall that the type tree index set for a pair (a,b) is $\{\varepsilon, 1, 2\}$ with the corresponding basic subterm set $\{(a,b), a, b\}$. Hence the projection π of (a,b) on 1 is $\pi_1 = (a,b)|_1 = a$.

Definition 5.3.11 (Basic Term θ-Join) *Let* $\theta : (\mathfrak{B}_\alpha \to \Omega) \to (\mathfrak{B}_\alpha \to \Omega) \to \Omega$ *be a predicate for some type* $\alpha \in \mathfrak{S}^c$. *If* A *and* B *are basic term relations with basic terms* $a|_i \subseteq \mathfrak{B}_\alpha$ *and* $b|_j \subseteq \mathfrak{B}_\alpha$ *respectively for some* $(i,j) \in I_A \times I_B$ *then the* basic term θ-join $\bowtie_{i\theta j}$ *of* A *and* B *is defined as*

$$A \bowtie_{i\theta j} B = \sigma_{i\theta j}(A \times B).$$

Following the traditional relational model, we also define a generalised form.

Definition 5.3.12 (Basic Term Generalised Join) *Let* φ *be a proposition that consists of atoms as allowed in basic term* θ*-restriction and the logical operations* $\wedge$, $\vee$ *and* $\neg$. *If* A *and* B *are basic term relations then the* basic term join $\bowtie_\varphi$ *is defined as*

$$A \bowtie_\varphi B = \sigma_\varphi(A \times B).$$

The closeness in form of the definition of the basic term join to that of the relational join facilitates the following result.

Proposition 5.3.13 *Relational joins are a special case of basic term relational joins.*

Proof 5.3.14 *Assume relation* $R \subseteq D_1 \times \cdots \times D_n$ *for some domains* $D_1, \ldots, D_n$. *Assume appropriate type constructors and data constructors such that* $D_1, \ldots, D_n \subseteq \mathfrak{B}$. *Let basic term relation* $S \subseteq D_1 \times \cdots \times D_n$. *Let* I_R *be the relation index of* R *and* I_S *be the basic term relation index of* S. *Clearly there is a surjection from* I_R *into* I_S *and thus from the set of tuple items in each tuple in* R *to the set of subterms in each corresponding basic term tuple in* S. *Assume the* θ *operators are available for basic terms and the result follows.*

5.3.4 Basic Term Relational Algebra

Given the basic term representation and fundamental basic term operations defined above we can now define the *basic term relational algebra* closely mirroring our earlier definition of the relational algebra.

Definition 5.3.15 (Basic Term Relational Algebra) *Let* $\mathfrak{D}$ *be a collection of non-empty domains* $\{\mathfrak{B}_\alpha\}_{\alpha \in \mathfrak{S}^c}$. *Let* R *be a set of basic term relations such that* $R =$

$\{R \mid R \subseteq \mathfrak{B}_\alpha, \alpha \in \mathfrak{S}^c \ \wedge \ \mathfrak{B}_\alpha \in \mathcal{D}\}$. *Let* ind *be a total function from basic term relations* $R \in \mathbf{R}$ *to basic term relation indexes* $I_R \subseteq \mathbb{N}^+$. *Let* Θ *be a set of predicates over domains in* $\mathcal{D}$, *and the logical operations* $\wedge$, $\vee$ *and* $\neg$ *over the booleans. Let* O *be the set of operations: basic term union* $\cup$, *basic term difference* $-$, *basic term product* $\times$, *basic term projection* π, *and basic term restriction* σ. *The* basic term relational algebra *over* $\mathcal{D}$, dom, $\mathbf{R}$, ind, Θ *and* O *is the 6-tuple* $\mathcal{R} = (\mathcal{D}, dom, \mathbf{R}, ind, \Theta, O)$. *An* algebraic expression *over* $\mathcal{R}$ *is any expression formed legally, according to the definitions of the operations, from the relations in* $\mathbf{R}$ *using the operations in* O.

5.4 Approximate Relational Joins

Having defined the basic term relational algebra as a formal framework we now introduce an extension to the basic term θ-join that moves beyond exact matches to enable approximate matches. In this section we introduce *approximate relational joins* as an abstract concept and in the following section we describe one possible approach to its concrete implementation.

For an exact relational join, the behaviour of the natural join in removing duplicate tuple items after joining on them is appropriate because the presence of duplicates adds no information to the data. However, in the context of an approximate relational join, the joined-on tuple items need not be identical because approximate equality is sufficient and so both values may hold useful information. For our intended profiling and matching application domain, we choose not to remove this information. Hence the following definition of *approximate relational joins* is based upon the θ-join rather than the natural join because the former does not remove tuple items.

5.4.1 Proximity-Joins

In order to turn an exact relational join into an approximate one it is necessary to replace the exact θ operator in θ-restriction with a suitable approximate version. For example, substituting exact equality $=$ with an approximate equality $\approx$ enables joining on tuple items that are either the same or in some way sufficiently similar.

One method of implementing approximate equality $\approx$ is to use a distance metric or pseudo-metric *dist*, defined on the domain of a pair of relational tuple items, together with a threshold δ to define a *proximity relation*.

Definition 5.4.1 (Proximity) *If the function* $dist : D \times D \to \mathbb{R}$ *is a distance on pairs of values from some domain* D *and* $\delta \in \mathbb{R} \ \ (\delta \geqslant 0)$ *is a threshold then* proximity *is a predicate* $\approx : D \times D \to \mathbb{B}$ *defined by*

$$\forall x, y \in D \ \ (x \approx y) \iff \{(x,y) \mid dist(x,y) \leqslant \delta \ \wedge \ x, y \in D\}.$$

By the definition of distance, the co-domain of *dist* is not constrained to have an upper bound. Some normalising function φ may be used to apply an upper bound to a distance. The function $\varphi : \mathbb{R} \to \mathbb{R}$ must be a non-decreasing function from the positive reals into some closed interval, typically $[0,1]$, such that $\varphi(0) = 0$, $\varphi(v) >$

0 if $v > 0$, and $\varphi(v+u) \leqslant \varphi(v) + \varphi(u)$, for each v and y. Example choices of φ from [Llo03] are $\varphi(v) = min(v,1)$ or $\varphi(v) = \frac{v}{v+1}$. Alternatively, the normalisation may be performed in the feature space of $x, y \in D$ so that $dist(x,y)$ is inherently normalised. For example, if the distance is derived from a kernel then a normalising kernel may be used [GLF04].

Proposition 5.4.2 *$\approx$ is a dependency relation but not necessarily an equivalence relation.*

Proof 5.4.3 *A dependency relation is reflexive and symmetric but not necessarily transitive; an equivalence relation is reflexive, symmetric and transitive.*

1. *If $\forall x \in D : dist(x,x) = 0$ then $\approx$ is reflexive over some domain D, which is true by dist being a pseudo-metric.*

2. *If $\forall x, y \in D : dist(x,y) = dist(y,x)$ then $\approx$ is symmetric over some domain D, which is true by dist being symmetric.*

3. *If $\forall x, y, z \in D : dist(x,y) \leqslant \delta \ \wedge \ dist(y,z) \leqslant \delta \ \Rightarrow \ dist(x,z) \leqslant \delta$ then $\approx$ is transitive over some domain D. This is trivially not always true. Assume $x, y, z \in \mathbb{R}$, $dist(x,y) = |x-y|$ and $\delta = 4$. If $x = 1$, $y = 3$ and $z = 6$ then $dist(1,3) \leqslant 4 \ \wedge \ dist(3,6) \leqslant 4 \ \nRightarrow \ dist(1,6) \leqslant 4$. Thus $\forall x, y, z \in D : dist(x,y) \leqslant \delta \ \wedge \ dist(y,z) \leqslant \delta \ \nRightarrow \ dist(x,z) \leqslant \delta$ and so $\approx$ is not necessarily transitive.*

Definition 5.4.4 (Proximity-Join) *Let $\approx : D \times D \rightarrow \mathbb{B}$ be a proximity for some domain D. If A and B are relations with tuple items $a|_i \in D$ and $b|_j \in D$ respectively for some $(i, j) \in I_A \times I_B$, the* proximity-join $\widetilde{\bowtie}_{i \approx j}$ *of A and B is*

$$A \,\widetilde{\bowtie}_{i \approx j}\, B = \sigma_{i \approx j}(A \times B).$$

The same historical notational convention is followed here for the subscripted $\approx$ as for the subscripted θ described earlier for the exact θ-join. The proximity-join as defined here is an approximate analogue of the exact relational equi-join. By choosing other proximity relations that are approximate analogues of exact relations, for example where $\theta \in \{=, \neq, <, \leqslant, >, \geqslant\}$, an approximate version of the relational θ-join might be defined. In this chapter we restrict our attention to the proximity-join.

5.4.2 Basic Term Proximity-Join

The choice of basic terms as our knowledge representational formalism means that in order to define a proximity-join on basic terms we require a distance metric or pseudo-metric *dist* that is defined on basic terms. In the next section we describe one possible choice for *dist* but here we leave its definition open.

Definition 5.4.5 (Basic Term Proximity-Join) *Let $\approx : \mathfrak{B}_\alpha \times \mathfrak{B}_\alpha \rightarrow \Omega$ be a proximity for some $\mathfrak{B}_\alpha$ of type α. If A and B are basic term relations with subterms*

$a|_i \in \mathfrak{B}_\alpha$ *and* $b|_j \in \mathfrak{B}_\alpha$ *respectively for some* $(i,j) \in I_A \times I_B$*, then the* proximity-join $\widetilde{\bowtie}_{i \approx j}$ *of* A *and* B *is defined as*

$$A \,\widetilde{\bowtie}_{i \approx j}\, B = \sigma_{i \approx j}(A \times B).$$

This definition closely parallels that of the approximate relational join on account of the following: the basic term relation is a set which allows the same set theoretic operators from the relational case to apply; the basic term relation index fulfills the same role as the relation index from the relational case; and, finally, the kernel for basic terms' own inductive definition implicitly handles the often recursive nature of structured data.

5.5 Proof of Concept

In this section we explore proof of concept definitions and demonstration implementations of one possible approach to approximate relational joins based on distances derived from kernels on basic terms.

5.5.1 Kernels and Distances for Approximate Joins

Kernel functions are an effective way of inducing distances on a wide variety of data structures [STC04]. In Chapter 2 we noted that a class of kernels, known as positive semi-definite kernels, induce pseudo-metric distances [GLF04] and that there is a straight forward way to turn any such kernel into a distance by Definition 2.3.24 (p.61). One particularly relevant kernel function for our chosen representation of structured data is the *default kernel for basic terms* [GLF04], as defined in Definition 2.3.20 (p.59) and described in "Section 2.2.4 – Distances for Structured Data" (p.46), but other kernels and distances may also be suitable. In Chapter 2 we gave a number of example calculations on different data structures. By way of recap and because it is important for the demonstrations in this chapter, below we give another example calculation of the default kernel, this time where the data structure is a sets of strings.

Example 5.5.1 (Default Kernel on Sets of Strings) *Let* S *be a nullary type constructor for strings and* $A,B,C,D : S$*. Choose* κ_S *and* κ_Ω *to be the matching kernel. Let* s *be the set* $\{A,B,C\} \in \mathfrak{B}_{S \to \Omega}$*,* $t = \{A,D\}$*, and* $u = \{B,C\}$*. Then*

$$\begin{aligned}
k(s,t) &= k(A,A)k(\top,\top) + k(A,D)k(\top,\top) + k(B,A)k(\top,\top) \\
&\quad + k(B,D)k(\top,\top) + k(C,A)k(\top,\top) + k(C,D)k(\top,\top) \\
&= \kappa_S(A,A)\kappa_\Omega(\top,\top) + \kappa_S(A,D)\kappa_\Omega(\top,\top) + \kappa_S(B,A)\kappa_\Omega(\top,\top) \\
&\quad + \kappa_S(B,D)\kappa_\Omega(\top,\top) + \kappa_S(C,A)\kappa_\Omega(\top,\top) + \kappa_S(C,D)\kappa_\Omega(\top,\top) \\
&= \kappa_S(A,A) + \kappa_S(A,D) + \kappa_S(B,A) + \kappa_S(B,D) \\
&\quad + \kappa_S(C,A) + \kappa_S(C,D) \\
&= 1 + 0 + 0 + 0 + 0 + 0 \\
&= 1.
\end{aligned}$$

Similarly, $k(s,u) = 2$ *and* $k(t,u) = 0$*.*

Continuing the sets of strings example, the following example illustrates the calculation of a distance from the default kernel for basic terms (using Definition 2.2.9 from p.46).

Example 5.5.2 (Default Distance on Sets of Strings) *Let* $s = \{A,B,C\}$, $t = \{A,D\}$, *and* $u = \{B,C\}$ *where* $s,t,u \in \mathfrak{B}_{S\to\Omega}$. *We have* $k(s,s) = 3$, $k(t,t) = 2$ *and* $k(u,u) = 2$. *Then,* $d_k(s,t) = \sqrt{3-(2\times 1)+2} = 1.73$, $d_k(s,u) = \sqrt{3-(2\times 2)+2} = 1$, *and* $d_k(t,u) = \sqrt{2-(2\times 0)+2} = 0$.

We next recap the intuition of the definitions of *kernels on data constructors* (Definition 2.3.18, p.58) and *support* (Definition 2.3.19, p.58) before presenting our own more detailed definition of the default kernel for basic terms, which we refer to as the *structurally weighted default kernel for basic terms*.

Kernels on data constructors: As with previously published definitions of the default kernel for basic terms, our definition assumes the existence of a default kernel on the data constructor of each atomic basic type. For example, a kernel on integers, a kernel on reals, a kernel on booleans, and so on. The same applies to data constructors for any domain-specific atomic types where the kernel may either be an alias for one of the aforementioned atomic basic types or a bespoke kernel function.

Support: Support is a function, supp, defined on basic abstractions. The support of a basic abstraction is its set of keys. For example, for a multiset $t \in \mathfrak{B}_{M\to\mathrm{Nat}}$ where $t = \langle (A,3),(B,1),(C,2)\rangle$ and $A,B,C : M$, then $\mathrm{supp}(t) = \{A,B,C\}$.

Here we restate the definition of the default kernel for basic terms introduced in [GLF04] to explicitly include kernel modifier functions. We found this version of the definition helpful when implementing the kernel for basic terms in Prolog for the demonstrations in this section. Our definition adds a higher-order parameter, $\kappa_{\mathrm{modifier}}(p)(k)$, which may be used to introduce domain-specific bias through weighting and smoothing applied to data instances of chosen types. Given a modifier and its parameters, p an element of the parameter space $\mathcal{P}$, and a kernel k, modifier function $\kappa_{\mathrm{modifier}} : \mathcal{P} \to (\mathcal{X}\times\mathcal{X}\to\mathbb{R}) \to (\mathcal{X}\times\mathcal{X}\to\mathbb{R})$ maps any kernel to the modified kernel. This behaviour was implicit in the description from [GLF04] but our restatement makes it explicit. As in the original, our definition below also assumes existence of a modifier kernel associated with each basic type.

Definition 5.5.3 (Structurally Weighted Default Kernel for Basic Terms) *Function* $k : \mathfrak{B}\times\mathfrak{B}\to\mathbb{R}$ *is defined inductively on the structure of terms in* $\mathfrak{B}$ *as follows.*

1. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = T\,\alpha_1 \ldots \alpha_k$, *for some* $T,\alpha_1,\ldots,\alpha_k$, *then*

$$\begin{aligned} k(s,t) &= \kappa_{modifier}(p)(k_\alpha)(s,t) \\ k_\alpha(s,t) &= \begin{cases} \kappa_T(C,D) & \text{if } C \neq D \\ \kappa_T(C,C) + \sum_{i=1}^{n} k(s_i,t_i) & \text{otherwise} \end{cases} \end{aligned}$$

where s *is* $C\,s_1 \ldots s_n$ *and* t *is* $D\,t_1 \ldots t_m$.

2. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = \beta \to \gamma$, *for some* β, γ, *then*

$$\begin{aligned} k(s,t) &= \kappa_{modifier}(p)(k_\alpha)(s,t) \\ k_\alpha(s,t) &= \sum_{\substack{u \in supp(s) \\ v \in supp(t)}} k(V(s\ u), V(t\ v)) \cdot k(u,v). \end{aligned}$$

3. *If* $s,t \in \mathfrak{B}_\alpha$, *where* $\alpha = \alpha_1 \times \cdots \times \alpha_n$, *for some* $\alpha_1, \ldots, \alpha_n$, *then*

$$\begin{aligned} k(s,t) &= \kappa_{modifier}(p)(k_\alpha)(s,t) \\ k_\alpha(s,t) &= \sum_{i=1}^{n} k(s_i,t_i), \end{aligned}$$

where s *is* $(s_1,\ldots,s_n)$ *and* t *is* $(t_1,\ldots,t_n)$.

4. *If there does not exist* $\alpha \in \mathfrak{S}^c$ *such that* $s,t \in \mathfrak{B}_\alpha$, *then* $k(s,t) = 0$.

The intuition for the meaning of the above definition is unchanged from our original explanation immediately following Definition 2.3.20 (p.59) except that in the new definition it is possible to see exactly where and when kernel modifiers are applied in the recursive evaluation process. As before, modifiers may be associated with types at any depth in the type structure. This naturally includes the type of the individuals being compared, i.e. the type of the individuals (e.g. the type of publications or the type of Programme Committee members from our submission sifting use case), and not just their subtypes. This is a natural consequence of individuals themselves being represented as basic terms, which will fall into part 1, 2 or 3 of the above definition, each of which includes a kernel modifier clause. In Example 5.5.4 we revisit our earlier example that defined a default kernel on sets of strings, Example 5.5.1, and extend it to include modifiers.

Example 5.5.4 (Applying a Kernel Modifier) *Let* S *be a nullary type constructor for strings and* $A,B,C,D : S$. *Let* $s,t,u \in \mathfrak{B}_{S\to\Omega}$ *where* $s = \{A,B,C\}$, $t = \{A,D\}$, *and* $u = \{B,C\}$. *Choose* κ_S *and* κ_Ω *to be the matching kernel. Choose* k *be the* default kernel on sets of strings *from Example 5.5.1 where it was shown that,* $k(s,t) = 1$, $k(s,u) = 2$ *and* $k(t,u) = 0$. *Choose* $\kappa_{normalised}(k)(x,x')$ *as the modifier for the type of sets of strings, recalling that,*

$$\kappa_{normalised}(k)(x,x') = \frac{k(x,x')}{\sqrt{k(x,x)k(x',x')}}.$$

The reflexive kernel values required by the modifier are $k(s,s) = 3$, $k(t,t) = 2$ *and*

$k(u,u) = 2$, *which correspond to the cardinality of the respective sets* s,t *and* u.

$$\begin{aligned}\kappa_{normalised}(k)(s,t) &= \frac{k(s,t)}{\sqrt{k(s,s)k(t,t)}} \\ &= \frac{1}{\sqrt{3\cdot 2}} \\ &= 0.4082\end{aligned}$$

Similarly, $\kappa_{normalised}(k)(s,u) = \frac{2}{\sqrt{3\cdot 2}} = 0.8165$ *and* $\kappa_{normalised}(k)(t,u) = \frac{0}{\sqrt{2\cdot 2}} = 0$.

Applying weighting schemes to data before, during or after its comparison is common practice in machine learning [Fla12] and is used to influence the relative importance of different components of the overall comparison. The elegance of the structurally weighted default kernel for basic terms is that it provides weighting at any depth in the type structure, not just on the top-most type of individuals, and gives the user a mechanism for engineering domain-specific weights by associating modifiers with semantically meaningful types. The disadvantage, in this or any other highly tunable comparison function, is that there are a lot of parameters to tune – so much so that it is not uncommon for these parameters to be learnt themselves using machine learning or optimisation techniques. An appealing characteristic of the default kernel for basic terms is that the structure and types used in the knowledge representation provide an intuitive and more principled approach to choosing such parameters manually.

One of the other strengths of the default kernel is that it allows any valid kernel to be associated with a specific type. For example, the following p-spectrum kernel, defined on strings, is used in our demonstrations in the next section.

Definition 5.5.5 (p-Spectrum Kernel [STC04]) *The feature space* F *associated with the* p*-spectrum kernel is indexed by* $I = \Sigma^p$*, with the explicit embedding from the space of all finite sequences over and alphabet* Σ *to a vector space* F *and is given by* $\phi^p_u(s) = |\{(v_1,v_2) : s = v_1 u v_2\}|, u \in \Sigma^p$*. The associated kernel is defined as* $\kappa_p(s,t) = \langle \phi^p(s), \phi^p(t)\rangle = \sum_{u\in\Sigma^p} \phi^p_u(s)\phi^p_u(t)$.

In other words, the p-spectrum kernel of a pair of strings is equivalent to the dot product of the multiplicity vectors of all their common substrings of length p. For example, for $p = 2$, $\kappa_p(\text{"fat cat"}, \text{"mat"})$ is,

$$\langle(\text{"fa"},1),(\textbf{"at"},2),(\text{"t}\textvisiblespace\text{"},1),(\text{"}\textvisiblespace\text{c"},1),(\text{"ca"},1)\rangle \cdot \langle(\text{"ma"},1),(\textbf{"at"},1)\rangle = 2\times 1 = 2.$$

When $p = 1$ this reduces to counting the number of characters the two strings have in common. e.g., for $p = 1$, $\kappa_p(\text{"fat cat"}, \text{"mat"}) = (2\times 1) + (2\times 1) = 4$.

5.5.2 Demonstrations

In this section we explore the potential utility of the *basic term relational algebra* (Definition 5.3.15) through the lens of the following proof of concept demonstrations.

D1. A kernel matrix component in a machine learning workflow.

D2. A replacement matching component for SubSift (Chapter 3).

In each case we implement the *basic term proximity-join* (Definition 5.4.5) with the kernel for basic terms as the underlying measure of proximity. Demonstrations D1 and D2 both share a common implementation of the join.

In keeping with the submission sifting use case and our earlier chapters, these proof of concept demonstrations are also set in the domain of academic publications. Heterogenous datasets within this domain include CORA, DBLP, Citeseer and Google Scholar. Interesting higher-order approximate joins between pairs (A, B) of these datasets might, for instance, include the following.

- $A \mathbin{\widetilde{\bowtie}}_{Author.name} B$, authors in A and B that have similar names
- $A \mathbin{\widetilde{\bowtie}}_{Author.affiliation} B$, authors in A and B affiliated to the same institution
- $A \mathbin{\widetilde{\bowtie}}_{Author} B$, authors in A and B similar across all their properties
- $A \mathbin{\widetilde{\bowtie}}_{Publication.venue} B$, publications in A and B from the same venue
- $A \mathbin{\widetilde{\bowtie}}_{Publication.coauthors} B$, publications in A and B with similar coauthors

Demonstration 1 – Kernel Matrix Component

We have implemented the higher-order relational projection, restriction and join operators and a range of supporting kernels, including the kernel for basic terms, in Prolog. Prolog does not natively support the data types of the higher-order logic necessary to represent basic terms. However, emulation of typed data, basic tuples, basic structures and basic abstractions (including sets and multisets) has proven to be unproblematic in practice. The representational approach taken was to annotate every data value with the name of its type using a functor `::`. Examples of some annotated atomic types are `42::nat`, `3.14::float`, `a::char`, `true::bool`, `'MIT Press'::string`. Example 5.5.6 shows how this annotation scheme is used to label more complex data types required to represent individuals in the application domain, e.g. publications and authors, as basic terms.

Example 5.5.6 (Representing a Basic Term as a Prolog Term) *The following prolog term is a type annotated representation of basic term describing an individual from the CORA dataset. Duplicates in the dataset are labelled as belonging to a class named after the first author of the represented paper, i.e. class* `fahlman1990b` *in this instance.*

```
fahlman1990b(
  tuple(
    abstraction(
      ('S.E. Fahlman'::string)::author-true::bool,
      ('C. Lebiere.'::string)::author-true::bool
    )::authors,
    ('The cascade - correlation architecture .'::string)::title,
    ('Advances in Neural Information Processing,'::string)::venue,
    (''::string)::year
  )::publication
)::class.
```

The types named `author`, `title`, `venue`, *and* `year` *are aliases for the type* `string`. *The type* `authors` *represents a basic abstraction* $\mathfrak{B}_{Author \to \Omega}$, *a set of* `author`.

Although annotating data values with their type is expensive in storage and the amount of pre-processing of data, subsequent evaluation of the kernel for basic terms becomes rather elegant and can closely follow Definition 5.5.3. Similarly, the type names provide a convenient way of declaratively associating the chosen kernel type with each data type. The latter point is why the type aliases depicted in Example 5.5.6 are not optimised out of the data in pre-processing; leaving type aliases such as `author`, which is an alias for `string`, allows a different kernel to be associated with that data value as opposed to, say, `venue` which is also an alias for `string` and without the aliases the two types could only be associated with the same kernel.

The Prolog code and data (as well as some supporting Matlab scripts) used in these demonstrations are available from the `prologkernels` subversion repository on Google Code[7].

This first demonstration of an approximate join on structured data is implemented as the following simple sequential machine learning workflow[8], which we will describe in further detail shortly.

1. Compute the pairwise kernel values between all items in the input relations and store the result as a kernel matrix.
2. Calculate a distance matrix from the kernel matix.
3. Apply agglomerative hierarchical clustering to the distance matrix to produce a list of clusters at montonically increasing distances.
4. Draw a dendrogram from the list of clusters to help inform decison making in Step 7 below.
5. Using supplied ground truths, count the number of pairs in each of the four quadrants of a confusion matrix (as discussed below).
6. Draw a precision-recall chart based on the supplied ground truths and the list of clusters.

[7] Code and data – `http://code.google.com/p/prologkernels/`, visited May 2014.

[8] The first step in the above sequence uses the Prolog matching component described ealier in this section; the remaining steps are implemented as Matlab scripts.

7. Select a point on the precision-recall chart as the threshold for the approximate join such that pairs in the same cluster at this threshold are treated as matches.

A number of distance-based methods could be used to implement the approximate join, including k-means, k-NN, and agglomerative hierarchical clustering. We chose the latter as our intuition was that the dendrogram might be useful in visualising and charactering the join, but were this not a proof of concept demonstration the other methods should also be evaluated and compared as part of the workflow. Our aim here is to show that a join could be implemented in this way; we leave to future work the task of cross-comparing different elements of the workflow, alternative parameter settings, visualisations and so on as they are not part of our Thesis.

Although this is a clustering method more normally associated with unsupervised learning, in this workflow we make use of the ground truth labelling to achieve a supervised setting so that a threshold for the join may be chosen. The intention is that to perform an approximate join on a previously unseen data based on this workflow one would use the same threshold on the assumption that the unseen data shares the same characteristics as this 'training' data. For simplicity we are manually selecting the threshold but a more complete and automated implementation of this workflow, for example, using RapidMiner (Weka) components could try out and evaluate different possible thresholds automatically.

The dendrogram is created to help manual selection of the threshold. It represents a progressive series of possible clusterings (i.e. possible joins), with instances in the same cluster being leaves of the same sub-tree. The distance value at each non-terminal node represents a potential threshold δ at which to 'cut' the tree and arrive at a set of clusters. δ is the threshold from the proximity predicate from Definition 5.4.5. To help evaluate the quality of the clustering at a given δ we consider whether each pair of instance data is correctly classified as being in the same cluster or in different clusters; in other words we evaluate a binary classification on all pairs of instances to determine if the two instances in a pair refer to the same publication or to different publications. A confusion matrix is then calculated in order to determine precision and recall for this specific value of δ. To characterise a proximity-join across a range of thresholds we vary δ across the length of the tree and plot a precision-recall chart.

For the sake of evaluation we require the ground truth V for each join to be evaluated, where $V \subseteq A \,\widetilde{\bowtie}\, B$ and, for the case where the individuals as terms represent publications, $V = \{(a,b) \mid a \in A \wedge b \in B \wedge a \text{ and } b \text{ are variants of the same publication}\}$. The goal is to reconstruct V as $V' = A \,\widetilde{\bowtie}_s\, B$ by choosing an appropriate s from the intersection of the basic subterm sets of A and B. In reality, V is not usually available for pairs of different datasets. For this reason we narrow down our initial demonstration to consider self-joins, $A \,\widetilde{\bowtie}_s\, A$, on a single data set $A = CORA$, for which ground truths are available [CM05]. In the following demonstration we take an alternative approach to deal with the lack of labelled data in most of our real-world use cases for the basic term proximity-join. The $CORA$ [9] dataset consists of bibliographic citations, hand-labelled with unique identifiers so that variant citations of the

[9] The specific CORA data set used is an aggregation of all three CORA-REFS citation matching datasets (*fahl-labeled*, *kibl-labeled*, and *utgo-labeled*). The raw CORA-REF files have numerous XML mark-up errors which we have manually corrected to enabled parsing.

same paper share the same identifier.

To represent the publications we choose the following type structure[10].

```
type Publications = Publication -> Bool;
type Publication = (Coauthors,Title,Venue,Year);
type Coauthors = Coauthor -> Bool;
type Coauthor = String;
type Title = String;
type Venue = String;
type Year = String;
```

Hence by representing CORA as a basic terms relation of type *Publications* , where $\mathfrak{B}_{Publications} \in \mathfrak{B}$, we are able to execute the following basic term proximity-joins:

- $CORA \; \widetilde{\bowtie}_{Publication.Title} \; CORA$, a self join on only the publication's *Title* subterm;
- $CORA \; \widetilde{\bowtie}_{Publication.Venue} \; CORA$, a self join on only the publication's *Venue* subterm;
- $CORA \; \widetilde{\bowtie}_{Publication.Coauthors} \; CORA$, a self join on only the publication's *Coauthors* subterm, which is in turn a set of *Coauthor* subterms;
- $CORA \; \widetilde{\bowtie}_{Publication} \; CORA$, a self join on the entire *Publication* term.

For each join, to keep results comparable on our precision-recall chart, we choose the p -spectrum(2) kernel for strings and accept the default kernels for all other types. For the previously mentioned reasons of relevance to our Thesis, we do not optimise the default kernel for basic terms by choosing weighting modifiers that, for example, might be used to encode the intuition that a year of publication is less discriminating than the title of a publication when aggregated into an overall kernel on publications. In principle, given that we have demonstrated that the kernel can practically be evaluated on structured data using our type-based indexing, tuning of the kernel could be achieved using established techniques from machine learning [GLF04, Fla12].

For each of these four joins we constructed a dendrogram such as Figure 5.7 and calculated the corresponding precision-recall chart in Figure 5.8. Note that the trivial reflexive pairs, i.e. cluster sizes of 1, are ignored in the plots as they convey no useful information here and so lines are not interpolated to the top left of the chart (recall=0, precision=1). As would intuitively be expected, joins on *Publication.Title* is generally a better discriminator of publications than *Publication.Coauthors* and *Publication.Venue* . However, the default kernel for basic terms clearly effectively aggregates the information contained in the subterms of *Publication* to outperform any single one of the three subterms taken in isolation. The only exception being *Publication.Title* , which sometimes outperforms its parent *Publication* above recall values greater than 0.9 [11].

[10] *Year* is string rather than a numeric type due to non-numeric characters in the data. Also, *Venue* is constructed as a concatenation of venue-related fields; CORA has no venue field.

[11] Which we suspect might be due to a group of similar trivial formatting typos in *Publication.Title* that are 'corrected' by the p-spectrum kernel, but without this having sufficient influence in the top-level *Publication* kernel. Tuning of the kernel using modifiers might be able to rectify this.

However, to conclude, there is no single best value of δ and so the choice of threshold would have to be chosen based on the relative importance of precision versus recall in the application domain. An alternative approach to the proximity-join that avoids this potential problem in finding an absolute threshold is used in the following demonstration.

Demonstration 2 – SubSift Matching Component

In this demonstration we take a more realistic scenario in which we wish to join $A \widetilde{\bowtie}_s B$, where $A \neq B$ and A, B are two other bibliographic datasets, ILPNet2[12] and DBLP-SUB [Reu06] respectively. Also, unlike the previous demonstration, there is no labelled ground truth mapping from basic term relation A to basic term relation B and so this is an unsupervised problem setting. The reasons for choosing these two datasets is that they included publications from a community of academics that were known to us. So, despite the lack of a suitable ground truth mapping, we were able to take a more qualitative approach in evaluating the results of the quantitative matches. The kernel definitions and publications data structure remain unchanged from the previous demonstration.

The workflow for this demonstration is a very much shortened version of the previous one.

1. Compute the pairwise kernel values between all items in the input relations and store the result as a kernel matrix.
2. Calculate a distance matrix from the kernel matix.
3. Feed the distance matrix through SubSift's report generator to produce a report listing the nearest K items in B for each person in A along with the distance figure for each paper listed.
4. Allow the user to adjust K and a threshold δ manually.

Figure 5.9 shows a screen shot of the distance data, with $K = 10$ and $\delta = 20$, displayed via the SubSift reporting module (in effect replacing the cosine similarity matrix with a distance matrix and reversing the ranking order so that low values rank highest, i.e. the opposite ranking of a similarity score). The resultant report includes metadata that would be left out is deployed web application but which is helpful for understanding the behaviour of the kernel. Below we give an extract of the text of a report for one author from ILPNet2, with $K = 5$ and δ unchanged. The first record lists an ILPNet2 publication (one of 1,041) and the subsequent 5 lines list the top ranking publications from the 2,796 publications in DBLP-SUB.

```
rank score   publication(ilpnet2, i404, ['P. Flach'],
               'A model of inductive reasoning',
               'Knowledge Representation and Reasoning un...', 1994).
1    5.19648 publication(dblp, d17113, ['Peter A. Flach'],
               'A Model of Inductive Reasoning.', 'Logic ...', 1992).
```

[12] Available from `http://code.google.com/p/prologkernels/`, visited May 2014.

```
2    8.24621 publication(dblp, d17127, ['Peter A. Flach'],
               'Second-order Inductive Learning.', 'AII', 1989).
3    8.3666  publication(dblp, d17123, ['Peter A. Flach'],
               'Predicate Invention in Inductive ...', 'ECML', 1993).
4    8.48528 publication(dblp, d17130, ['Peter A. Flach'],
               'Towards a Theory of Inductive Lo...', 'ISMIS', 1991).
5    8.59146 publication(dblp, d17125, ['Peter A. Flach'],
               'Rationality Postulates for Induction', 'TARK', 1996).
```

This approach produces a report for every author and so the use cases for which this join would be useful are closer to submission sifting than to, for example, dataspaces where the requirement would be for an SQL-like join over a large dataset rather than this qualitative individual approach here. For the authors we recognise, this approach looks promising (for the subset of proximity-join applications where this presentation might be useful) although it is not possible without known mappings to make any overall assessment of the quality of the matches.

5.6 Summary

In this chapter we focused on the data comparison aspects of our uses cases, exploring whether *matching* of heterogeneous structured data could be expressed as declarative queries along similar lines to SQL queries in the relational model, rather than through the imperative domain-specific language used in our higher-order dataflow model from Chapter 4. The relational model and its associated relational algebra provide a formalism for querying relational data in SQL databases. Hitherto there has been no analogous model and algebra for querying structured data originating from heterogeneous sources. This is important in the context of this Thesis because structured data is prevalent across our use cases and the research intelligence domain; it is also an important problem in the wider context of web services where relational databases are widely used in their implementation and relational queries (e.g. SQL `select` statements) are often accepted as parameters to their SOAP or REST API methods.

To address this lack of a relational formalism for querying structured data we introduced a higher-order relational model, lifting the traditional relational representation to the basic terms in a higher-order logic that is better suited to the representation of structured data. A relational algebra for basic terms was defined as a single coherent formalism for querying heterogeneous structured data. This was achieved in two stages: first we upgraded the relational representation to a higher-order representation by substituting relational sets for basic terms; we then lifted the relational algebra's operators to this same higher-order formalism. We suggest that this algebra has the potential to bring the benefits of relational query languages to applications involving structured data – without the usual obfuscating transformation to a relational representation that would otherwise be required.

This higher-order relational algebra allowed us to explore whether the matching of structured data from the research intelligence domain could be usefully expressed as higher-order relational queries. The algebra introduced in this chapter includes a higher-order join operator which we showed to be a generalisation of the join operator from the traditional relational model. Our approach to implementing matching in

the higher-order relational algebra was to relax the join operator to be an approximate join and then to use that as the basis for matching structured data in our research intelligence domain. We demonstrated that a family of kernels and distances defined on basic terms can be used as the data comparison mechanism for implementing matching as an approximate join on structured data. These demonstrations also suggest that it may be possible to produce useful applications based on such a join in the research intelligence domain by evaluating example approximate queries on bibliographic datasets, joining on types ranging from sets of co-authors through to entire publications.

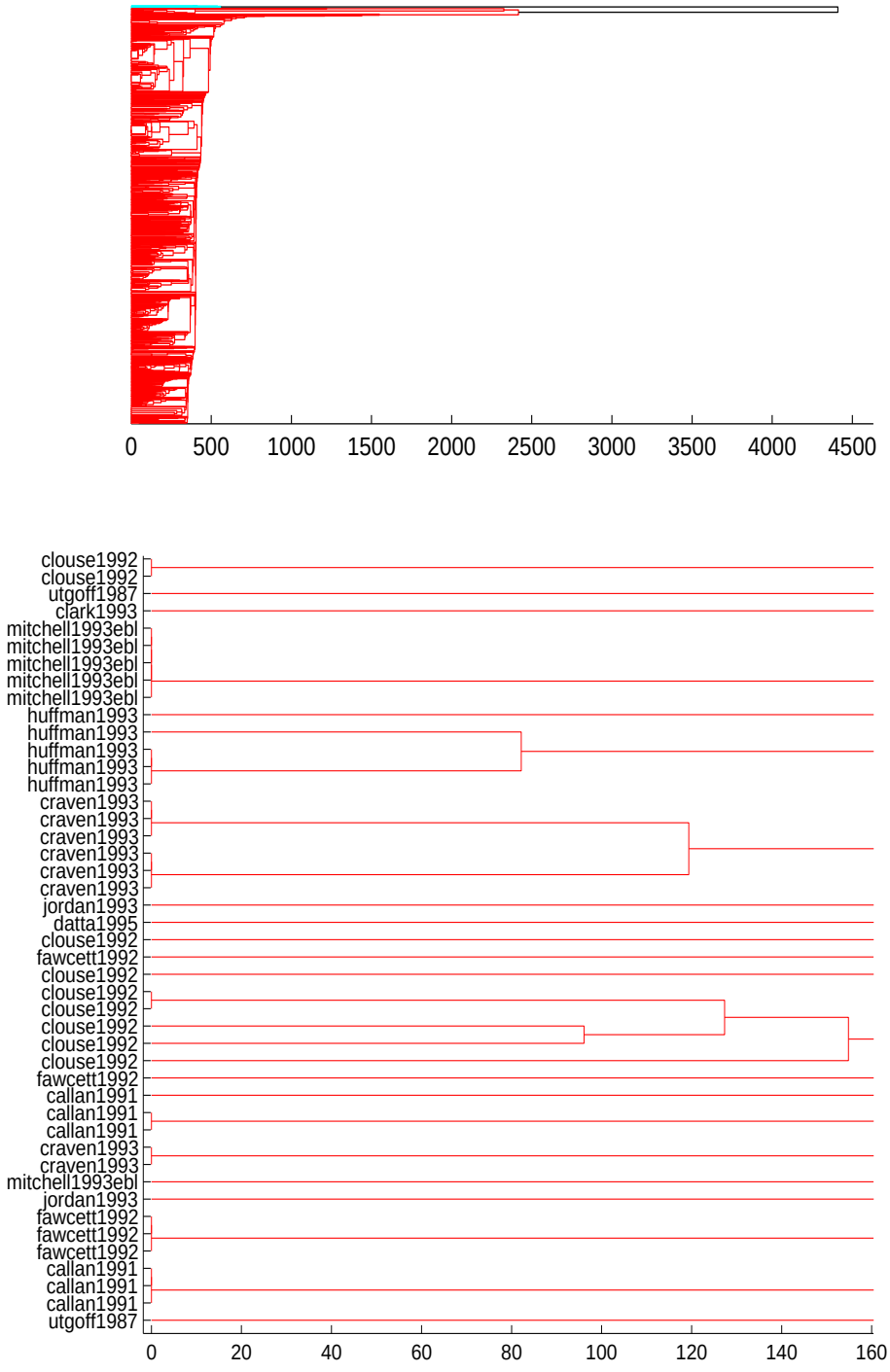


Figure 5.7: Dendrogram (above), showing clusterings at successive thresholds for a proximity-joins on the CORA publication type, and (below) a close-up showing labelled ground truths.

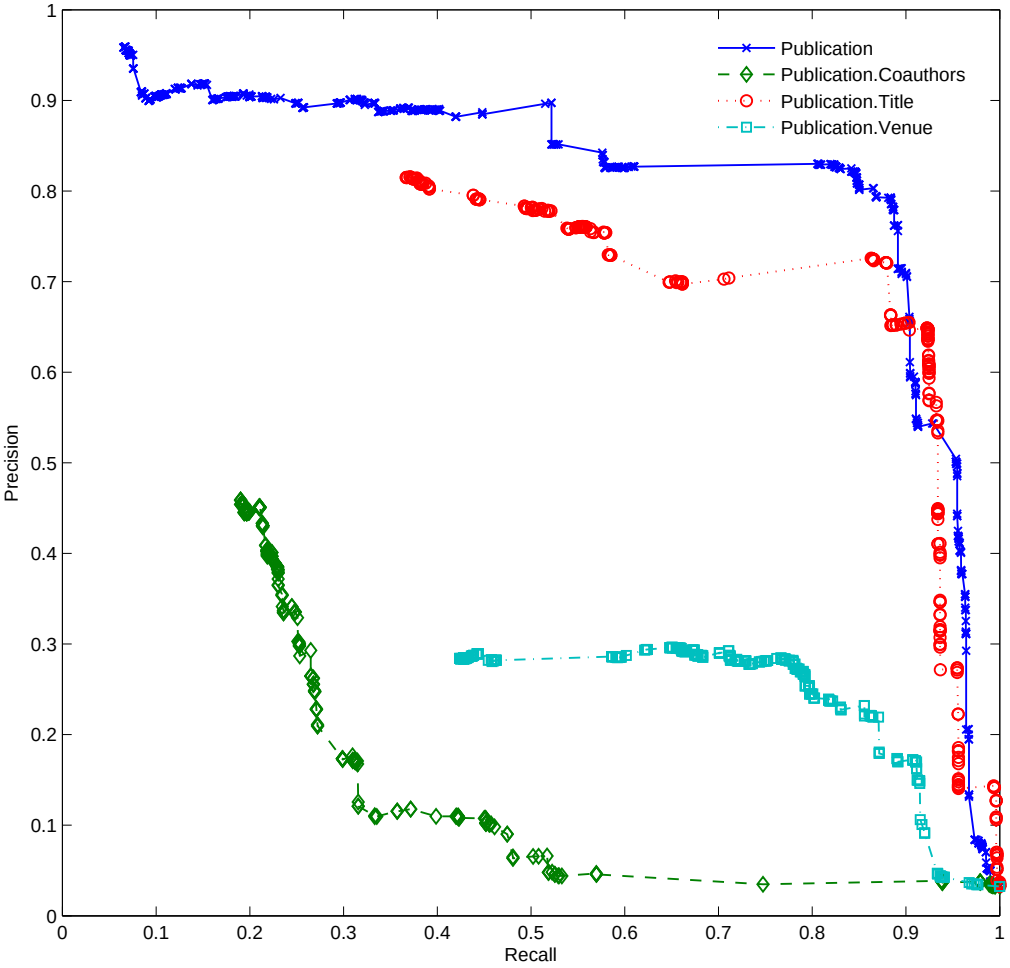


Figure 5.8: Precision and recall for various decompositions of the CORA publication type: *Publication, Coauthors, Title* and *Venue*.

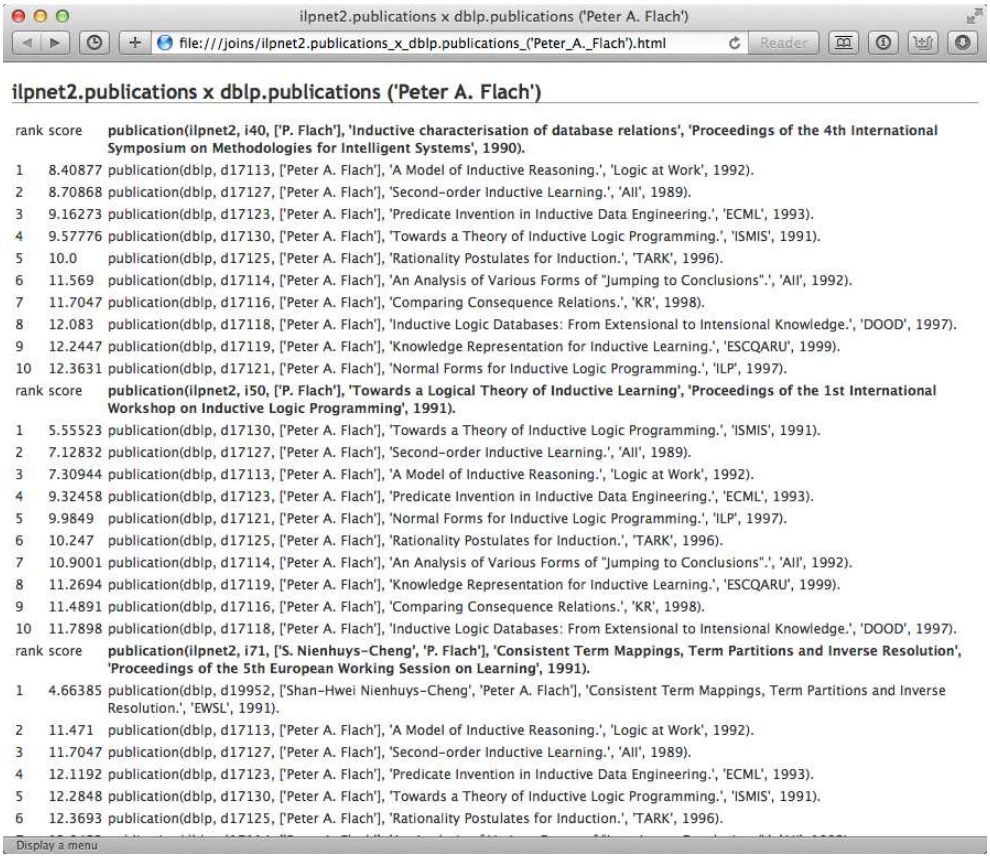


Figure 5.9: Basic term proximity-join as a SubSift report. Publications in bold are from ILPNet2 and the list of publications beneath are from DBLP-SUB.

Chapter 6

Related Work

In this chapter we survey existing systems that cover parts of the problem of comparing heterogeneous data and contrast these with our own work. Prior work relating to our motivational use cases is also reviewed and related to our approach from an engineering perspective. The chapter is organised into three main sections. In Section 6.1 we survey a broad range of topics in order to position our frameworks relative to those of related areas. In the more narrowly focused Section 6.2 we review the literature that led to our choice of algorithms and approach to profiling and matching, as well as to our rejection of some candidate alternative approaches. Section 6.3 compares our frameworks to previous work on the formalisation of dataflow and workflow, and with previous implementations of similar frameworks. In Section 6.4, we briefly mention some prior work that falls outside of the scope of this Thesis that, although necessarily drawn upon in our own work, is not itself relevant to our research objectives from Chapter 1.

6.1 Broadly Related Areas

The frameworks investigated in this Thesis are related to work in each of the following areas (*listed in alphabetical order*).

- Big Data
- Clustering
- Expert Finding
- Higher-Order Frameworks
- Scientometrics and Bibliometrics
- Semantic Web
- Text Processing
- Workflow Systems

We begin this section of our survey by discussing Big Data, which as well as having its own section also appears under some of the other headings.

6.1.1 Big Data

Large datasets arising from web crawls or data from high-throughput scientific instruments now exceed the capacity of current relational database implementations. Such data is known as *Big Data* and is typically characterised by its volume, velocity and variety [RU11].

In this Thesis, the heterogeneity of datasets in the domain is a central concern and so it is the *Big Variety* flavour of Big Data that is the most relevant of these, so called, Big Vs. However, in "Section 4.3.3 – Parallelisability and Scalability of the Model" we showed that pure transformations in our higher-order dataflow model satisfy the conditions of an *embarrassingly parallel* function and hence, in principle at least, are highly parallelisable [Fos95, WA99]. Proving that pure transformation is embarrassingly parallel does not guarantee than an implementation of our higher-order dataflow model will be highly parallelisable but it does strongly suggest that parallel implementation of the model is theoretically possible. Neither does it guarantee that an implementation will he highly scalable in the size of data that can be transformed but again it does suggest that scalable implementation is theoretically possible.

JSONMatch, our proof of concept implementation of the model, demonstrates that pure transformations can be computed one result item $V(c\,k)$ at a time such that only the inputs to a single item's computation need to be loaded into working memory at once. This shows that working memory requirements for pure transformation can indeed be made independent of the number of items in input and output relations. One of the other implications of this one-at-a-time computation of output items is that if executed serially, the model requires very little state information in order to record the current program control state during the execution, making suspension and resumption cheap, and so even our proof of concept implementation is able to process datasets that exceed the size of working memory on a standard shared-use web server – given enough time. For further details of the mechanism enabling this, see "Section 4.3.4 – Continuations in the Model" (p.121).

Dataflow approaches to analysing Big Data, using tools such as Pig, Hive and Hadoop, focus on batch processing of large datasets using parallelisation to reduce elapsed time to results, but at the expense of increased working storage, compute, development and operational costs. By contrast, in our JSONMatch dataflow framework, the speed to results is less important than the ease of integration with interactive web applications and web service mash-ups. In this sense JSONMatch occupies a niche of the wider Big Data domain. In principle, by virtue of the underlying model, JSONMatch could admit many of the same highly parallelisable design patterns used in MapReduce algorithms, large (but not massive) datasets may still be analysed on modest single-node hardware. For smaller datasets, non-parallelisable design patterns, such as shared global counters or accumulators, may be used to reduce the number of transformations in a dataflow or for algorithmic simplicity.

Higher-order dataflows implemented in Pig Latin would achieve Big Volume Big Data capability but, without developing a containing web service framework similar to our own, would lack the ease of use on Big Variety small datasets afforded by our JSONMatch implementation. Pig Latin is a high-level imperative dataflow language with extensibility through Java and arbitrary other languages such as Python, Perl, C/C++, etc. JSONMatch runs as a web service, with an inbuilt web-addressable

NoSQL store; it allows developers to specify and execute new dataflow transformations through REST API calls. For small data, the results of transformations are returned immediately; for larger data, HTTP polling detects completion. Extensibility is enabled through JSONMatch's ability to incorporate web service calls to external web services as embedded functions in higher-order parameters. One of the significant advantages of the JSONMatch framework over conventional Big Data systems is that web applications can compose workflows at runtime by specifying them as higher-order parameters to a REST API method.

NoSQL databases like CouchDB and MongoDB share similar JSON document storage and concurrent access features to JSONMatch, but both have sophisticated sharding features and built-in MapReduce support that is absent from our current implementation. CouchDB is most similar, having a built-in web server and REST API, and could be potentially be used to implement our higher-order dataflow model. However, CouchDB lacks support for the data mining and machine learning features of JSONMatch. Such features are found in analytics systems like RapidMiner and Datameer, both of which support dataflows with more complicated topologies than JSONMatch – although, arguably, with JSONMatch the complexity has just been moved into the embedded functions. However, it would be possible to incorporate REST calls from CounchDB to the Weka machine learning or the GATE text mining toolkits in order to gain analytics closer to those of RapidMiner and Datameer[1].

6.1.2 Clustering

Clustering data using similarity and distances is an active research topic in information retrieval and data mining [New01, GLF04, STC04] and has been applied before in e-Science workflows – for instance, a recent system automatically clusters search results according to the similarity of document content [ATT + 10]. However, SubSift and JSONMatch expose their functionality through web services rather than locking it within a single application or experimental framework. This does not preclude the use of SubSift to produce the input data to externally implemented clustering algorithms. For example, in Chapter 7 we report on exploratory work that uses SiftSift-computed similarity scores as the basis for clustering descriptions of members of a university department [PFS10c].

6.1.3 Expert Finding

Our description of "Use Case 2 – Finding an Expert" (p.3) has already introduced and relates directly to the long-established field of Expert Finding spanning both the research intelligence and business intelligence domains [YsK03, BAdR06, FZ07, MM07, DKL08]. This Thesis does not claim contributions to this field *per se* and instead focuses on frameworks for describing and building a broader range of applications in the research intelligence domain.

Over the years since the first Text Retrieval Conference (TREC) in 1992 [Har93], the task of finding experts on a particular topic has featured regularly in this long-running conference series and is now an active subfield of the broader text informa-

[1] It may also be possible to link CouchDB to these tools directly without using HTTP protocols.

tion retrieval discipline. There is also a degree of overlap with the fields of scientometrics and bibliometrics discussed later in this chapter, although expert finding tends to be more content-based (i.e. based on the text of documents) than link-based (i.e. based on cross-references between documents) as we discuss in that section.

In [YsK03] a distinction is made between external and internal (to an organisation) expert finding. There is a similar distinction in our own work where use case 2, as used to find reviewers for a journal paper across a research community, has mostly bee applied in the external setting whereas our additional variant use cases based on an effort to mine and map the University of Bristol's research landscape has a distinctly internal focus. These two settings each require different approaches to acquiring the data from which profiles are computed, with internal also having privileged access to corporate databases as a source of readily available data to simplify the matching task itself (although, of course, accessing corporate data can entail additional work in itself).

Expert finding systems have been developed commercially, including 'Agentware Knowledge Server' from Autonomy[2] and 'KnowledgeMail' from Tacit Knowledge Systems[3], the latter generating profiles from both email and word processor documents. Their internal algorithms are not published but they operate on the same content-based models from the literature and so it seems reasonable to assume their methods are also similar to published ones.

As we mentioned in our original description of the use case, expert finding can be viewed as a special case of the submission sifting use case but where the emphasis is on finding the best match for a single submission rather than for multiple submissions. This observation is reinforced by the fact that expert finding tends to use either the same text matching or information retrieval approaches for profiling and matching, as discussed later in this chapter.

6.1.4 Higher-Order Frameworks

Our higher-order dataflow model could be viewed as a combination of data- and task-parallel skeletons from higher-order algorithmic skeleton frameworks (ASkF) [GVL10]. JSONMatch resembles an ASkF wrapped in web services, with skeletons expressed in JSON which, unlike most ASkF, has the advantage of already being familiar to web developers – many of whom will not have a formal background in Computer Science and, based on our own experiences, it is not uncommon for such Web developers to only know the JavaScript language.

MDX queries for OLAP data warehouses have similarities to our higher-order data constructors, but have very different data models and are tightly coupled to the SQL relational model. However, it should be noted that we propose the higher-order relational representation solely as an approach for data integration tasks, not as a replacement for general purpose relational databases. Our present implementation certainly has none of the optimisations of a modern relational database. Ultimately though, a higher-order view could be layered on top of a traditional relational database system, efficiently combining the two approaches, so that higher-

[2] Autonomy – `http://www.autonomy.com`, visited May 2014.

[3] Since acquired by Oracle in 2008.

order queries are automatically translated into and executed as equivalent relational queries.

6.1.5 Scientometrics and Bibliometrics

Our applications and use cases sit within the domain of 'research intelligence' and much of the data discussed in this Thesis concerns profiling and matching the publications of researchers. Although, the theoretical and software frameworks introduced in previous chapters are not restricted to this domain; indeed, our higher-order dataflows and relational algebra have no particular ties to their origins in submission sifting and could potentially be applied in other domains. Notwithstanding this, it would be remiss not to mention related work in the fields of *bibliometrics* and *scientometrics* (themselves closely related) here.

- *Bibliometrics* is the quantitative analysis of academic publications and other research-related literature [P+69, Har09].
- *Scientometrics* does the same for scientific research and extends the measures to include patents, discoveries, data outputs and, in the UK, more abstract concepts such as 'impact' [BBB+13].

Publications from these research communities are concentrated in the journals "Scientometrics" and "Journal of the American Society for Information Science and Technology" although the origins of the disciplines lie in earlier statistical journals from as far back as the 1920s. Even though by comparison with bibliometrics, scientometrics encompasses additional measures, in practice the dominant approach in both domains is citation analysis of academic literature. Citation analysis measures the properties of networks of citation amongst publications and has much in common with hyperlink analysis on the web, where these measures employ similar graph theoretic methods designed to model reputation, with notable examples including 'Hubs and Authorities' [Kle99] and PageRank [BP98]. This contrasts with the content-based profiling and matching used in this Thesis, where we analyse the text of publications and web pages rather than their explicit inter-relationships. Content analysis is an active area of bibliometrics in particular and has been used in combination with with citation properties to link research topics to specific authors [RZGSS04]. The techniques used to analyse content in bibliometrics and scientometrics are the same as those described for profiling and matching later in this chapter. However, in principle, submission sifting could (and perhaps should) be extended to incorporate citation analysis.

6.1.6 Semantic Web

In "Section 2.2.1 – Graph-based Representations" we gave an introduction to the data representational formats used of the Semantic Web. Our goal of integrating and querying heterogeneous data is also a goal shared by the Semantic Web community [BLHL01]. The fundamental data model of the Semantic Web is the directed labelled graph, represented as RDF triples, which may be queried using the SPARQL query language [PS05]. Data structures such as lists, sets, multisets, trees and graphs are

readily supported through RDF Schema and the OWL ontology language [MvH12] and as such have similar representational advantages to basic terms as compared to the relational model. SPARQL queries can be used to retrieve a subgraph describing an individual that is analogous to a representation of that individual as a basic term. Conversely, it is straight forward to transform the same subgraph into a basic term in order to apply our own approach to RDF data. For RDF data integration, or *smushing* as it is informally known, the emphasis in the Semantic Web languages to-date has been on exact matching, using inverse-functional properties such as email addresses, homepage URLs or entity URIs. This is an obvious shortcoming in the presence of noisy data or representational variations between data from different sources. As will be discussed in much greater depth later in this chapter, to address the consequent data integration problem, work has been done in the area of ontology matching, including work on measuring proximity between ontologies [MS02].

Our approximate matching work differs from this explicit semantic integration approach in that we rely primarily on the implicit semantics of the type structure and data instances themselves. This is an advantage in cases where detailed ontological information is not available but potentially a disadvantage in other cases because background knowledge encoded in an ontology is not exploited in our approximate joins (Chapter 5). The incorporation of background knowledge into our approximate joins is an area for future work.

6.1.7 Text Processing

Numerous text processing web services exist for specific tasks within a workflow, particularly domain-specific ones. Examples include the long-standing TerMine service for the recognition of multi-word terms, AcroMine for the expansion of bioinformatic abbreviations, and numerous others listed in the BioCatalogue life sciences web services directory [FAM00, OA06, BTN+10]. However, SubSift services provide a service-based framework for managing and comparing collections of documents and for managing and publishing the results of analyses. To achieve the same with standalone text processing services would require the implementation of considerably more complex workflows.

Text mining workflows incorporating web services can be constructed through U-Compare's ability to embed Unstructured Information Management Architecture (UIMA) components within Taverna workflows, but this is not itself a web service [KDN+10]. Recently, an information retrieval approach has been applied to web service discovery where the same vector space model as is used in SubSift has been shown to be effective as one of the components in ranking the similarity of web service descriptions [HZC10]. Earlier work used similar techniques to construct semantic profiles of academics as the basis of a paper recommender system [ZL04b]. We recommend [BOHG13] for an up-to-date and wide ranging survey of recommender systems.

Such systems differ from SubSift in that their functionality is not general-purpose. "Use Case 1 – Submission Sifting" and "Use Case 2 – Finding an Expert" workflows, using SubSift as described in Chapter 3, could be used to supply the text matching component of a web service discovery application or paper recommender system respectively. Tigher integration of recommender systems into these workflows could

be achieved through JSONMatch where it would be possible to add a library of embedded functions for use within higher-order templates.

6.2 Approaches to Profiling and Matching

In "Section 2.1 – Terminology" (p.27) we defined the meaning of the terms *profiling* and *matching* in the context of this Thesis (Definitions 2.1.5 and 2.1.6 respectively). Although this Thesis is not especially concerned with particular algorithms for profiling and matching, it is important that the algorithms used in our proof of concept frameworks are known to perform sufficiently well in similar domains to our own. For this reason, in our main submission sifting use case we are initially interested in finding a "tried and trusted" algorithm for profiling and matching predominantly textual descriptions of people and of documents. That said, in later chapters we shift the focus away from purely textual data to include all forms of structured data. Therefore, we also survey some potentially relevant approaches to describing and integrating structured data.

Prior work of relevance for profiling and matching descriptions of people, database records and documents spans a range of traditional research fields. Over the following sections we review the following most relevant topics from these fields.

- Database Deduplication
- Record Linkage
- Information Retrieval
- Schema and Ontology Matching
- Dataspaces

For reasons that are explained in the following section, "6.4 – Scope", we do not explicitly review prior work in paper allocation, bibliometrics, citation matching and expert recommendation *per se*. For now we just note that the relevant content-based (also known as *language model* based) methods used in such papers draw upon the same topics reviewed below – particularly information retrieval. Illustrative examples include [GBL98, BHCNm99, LBG99, BL01, PMM+03, DKL08, CZB11], although we highlight that such work also employs a host of other methods in combination with content-based methods, but these are not relevant to our research questions and so are not touched upon here. However, in the Future Work chapter of this Thesis we suggest how alternative methods could be incorporated into our own frameworks.

6.2.1 Schema and Ontology Matching

We have already mentioned *schemas* in our discussion of knowledge representation in "Chapter 2 – Background". In this section we revisit schemas and also discuss *ontologies*, their more general form, as we jointly review the literature of both schema matching and ontology matching as they relate to profiling and matching. At times

the two bodies of literature overlap with each other and with a number of related research areas such as schema/ontology learning, schema/ontology merging, database integration, lexical integration and model management. Schema/ontology matching is also referred to in the literature as schema/ontology alignment, mapping, translation, integration and fusion but we will consistently use the terms schema/ontology matching.

The word *schema* is derived from the Greek word for shape. A schema, in the context of knowledge representation, is a data model expressed in some formal language. Perhaps the most familiar example is the database schema which describes the structure, content and, to a limited degree, the semantics of a database. Another example occurs in semi-structured data where XML Document Type Definitions (DTD) [BPSM$^+$06] and XML Schema [BM04] are used with the same descriptive purpose. The need for schema matching arises whenever data sources with different schema are merged or cross-queried. Finding a match between a pair of schema involves creating a mapping between semantically corresponding elements in each data source. This mapping may be one-one, many-to-one or many-to-many and may involve transformation of the data. In relational databases, this amounts to mapping fields in one database to fields in another. For example, a pair of schema describing relational data sources Δ_P and Δ_Q, which are paper author contact details and customer contact details respectively, may be represented as follows.

```
schema( schemaP,
  contact_details(author, institution_address)
)
schema( schemaQ,
  contact_details(name, street, town, city, postcode, country)
)
```

Then one possible output from a schema matching tool might identify the following mapping described below using the *same_as* relation to represent equivalence and *concat* to represent string concatenation.

```
mapping( schemaP, schemaQ,
  same_as(contact_details, author,
    contact_details, name),
  same_as(contact_details, institution_address,
    contact_details, concat(street, town, city, postcode, country))
)
```

Real-world mappings need to contain more information than is given above - for instance, data types (e.g. length limited strings versus unlimited), character encoding (e.g. ANSI versus UTF-8 - important for non-latin characters), normalisation (e.g. “Price, S." versus “S. Price"). Also, for illustrative purposes, identifiers and multiple relations are not used in the above example but an efficiently normalised relational database would typically use both. Clearly, schema matching is a non-trivial problem. We will later review the various approaches to the problem from the literature and discover that automatic schema matching is, as has been concluded elsewhere

[RB01], still an open problem in the general case. Schema matching is, however, only a simpler sub-problem of the more general ontology matching problem.

The word *ontology* is derived from the Greek for the study of being - of things that exist or may exist in some domain. An ontology, in the context of knowledge representation, is a data model expressed in some formal language that describes a domain and can be used to reason about the objects in that domain. The primary difference between a schema and an ontology is that the former is designed to describe a data source whereas the latter is designed for describing data across multiple, potentially heterogeneous, data sources. In one sense, the intention of an ontology is to avoid the need for schema/ontology matching by creating a shared knowledge representation and enabling reasoning about extensions to that representation based on the shared core.

There are many differing formal definitions of *ontology* in the literature but the following, and its associated definition of *lexicon*, both due to [Mae02], incorporate the most common features. The generality of this definition makes it possible to trivially express schema, dictionary, thesaurus and taxonomy (also known as classification hierarchy or directory) structures as ontologies.

Definition 6.2.1 (Ontology [Mae02]) *An ontology structure* $\mathcal{O}$ *is a 5-tuple* $\mathcal{O} := (\mathcal{C}, \mathcal{R}, \mathcal{H}^{\mathcal{C}}, rel, \mathcal{A}^{\mathcal{O}})$*, consisting of*

- *two disjoint sets* $\mathcal{C}$ *and* $\mathcal{R}$ *whose elements are called* **concepts** *and* **relations** *respectively*
- *a* **concept hierarchy** $\mathcal{H}^{\mathcal{C}} : \mathcal{H}^{\mathcal{C}}$ *is a directed relation* $\mathcal{H}^{\mathcal{C}} \subseteq \mathcal{C} \times \mathcal{C}$ *which is called a concept hierarchy or taxonomy.* $\mathcal{H}^{\mathcal{C}}(C_1, C_2)$ *means that* C_1 *is a sub-concept of* C_2
- *a* **function** $rel : \mathcal{R} \to \mathcal{C} \times \mathcal{C}$*, that relates concepts non-taxonomically. For* $rel(R) = (C_1, C_2)$ *one may also write* $R(C_1, C_2)$
- *a set of ontology* **axioms** $\mathcal{A}_{\mathcal{O}}$*, expressed in an appropriate logical language. e.g. first order logic*

Definition 6.2.2 (Lexicon [Mae02]) *A lexicon for the ontology structure* $\mathcal{O} := (\mathcal{C}, \mathcal{R}, \mathcal{H}^{\mathcal{C}}, rel, \mathcal{A}^{\mathcal{O}})$ *is a 4-tuple* $\mathcal{L} := (\mathcal{L}^{\mathcal{C}}, \mathcal{L}^{\mathcal{R}}, \mathcal{F}, \mathcal{G})$*, consisting of*

- *two sets* $\mathcal{L}^{\mathcal{C}}$ *and* $\mathcal{L}^{\mathcal{R}}$ *whose elements are called* **lexical entries** *concepts and relations respectively*
- *two relations* $\mathcal{F} \subseteq \mathcal{L}^{\mathcal{C}} \times \mathcal{C}$ *and* $\mathcal{G} \subseteq \mathcal{L}^{\mathcal{R}} \times \mathcal{R}$ *called* **references** *for concepts and relations respectively. Based on* $\mathcal{F}$*, let for* $L \in \mathcal{L}^{\mathcal{C}}$*,*

$$\mathcal{F}(L) = \{C \in \mathcal{C} \mid (L, C) \in \mathcal{F}\}$$

and for

$$\mathcal{F}^{-1}(C) = \{L \in \mathcal{L}^{\mathcal{C}} \mid (L, C) \in \mathcal{F}\}.$$

$\mathcal{G}$ *and* $\mathcal{G}^{-1}$ *are defined analogously.*

In general, one lexical entry may refer to several concepts or relations, and one concept or relation may be referred to by several lexical entries. An ontology structure with a lexicon is a pair $(\mathcal{O}, \mathcal{L})$*, where* $\mathcal{O}$ *is an ontology structure and* $\mathcal{L}$ *is a lexicon.*

Inevitably, any ontology must always be a balance between generality and specificity. The ideal ontology for a given domain is highly specific to that domain and will therefore be less applicable in other domains. By contrast, a general ontology may be widely applicable but is unable to capture all the fine details of a specific domain. Consequently, there is a natural tendency for the proliferation of domain-specific and application-specific ontologies. This in turn, despite the intention and aspirations of some ontology authors, gives rise to a strong requirement for ontology matching in almost any data integration scenario involving an ontology. Matching can sometimes be required within the same ontology because the expressiveness of ontology languages tends to give users many degrees of freedom in describing the same entities and their relations. Even in case of a single data source and a single taxonomy (a taxonomy is one of the simplest forms of an ontology), human individuality will lead to different classification labels being assigned to the same data instance by different users and in reverse, different experts will produce different ontologies for labelling the same data.

Attempts have been made to automatically match taxonomies, schemas and ontologies based only on their own internal structure without reference to instances of data labelled using that structure [RB01, Noy04]. These, so called, *shallow matching* approaches rely almost entirely on the availability of human-readable labels assigned to concepts. For example, the concept *person* in one ontology might be mapped to the concept *employee* in another. Similarly, considering a concept as its general down to specific path through a taxonomy might result in an attempt to map Google Directory's */arts/music/styles* to Yahoo Directory's */entertainment/music/genres*. Mapping between the lexical and taxonomic layers of an ontology are an essential part of measuring the overall similarity between ontological structures which is an important topic in ontology engineering for the task of retrieving relevant ontologies from a database of standard ontologies. As an illustration, in [MS02] the following similarity measures are used to build up an overall similarity measure. String matching (SM) based on Levenshtein's edit distance (ed) is used on left substrings of concept labels. SM is similarity between 0 (zero or bad match) and 1 (perfect or good match).

$$SM(L_i, L_j) := max\left(0, \frac{min(|L_i|, |L_j|) - ed(L_i, L_j)}{min(|L_i|, |L_j|)}\right) \in [0, 1].$$

Concept string matching is combined for n-tuples of concept labels to compare lexical consistency of two ontologies.

$$SM(\vec{l}_i, \vec{l}_j) := \sqrt[n]{\Pi_{m:=1\ldots n} SM(l_i^m, l_j^m)} \in [0, 1].$$

This measure is then averaged as an asymmetric measure of the extent of ontological overlap ($\overline{SM}$).

$$\overline{SM}(\mathcal{L}_i,\mathcal{L}_j) := \frac{1}{|\mathcal{L}_1|}\sum_{L_i\in\mathcal{L}_1} max_{L_j\in\mathcal{L}_2} SM(L_i,L_j).$$

At the taxonomic level, the conceptual cotopy (CC) of a concept, i.e. the set of all its super- and subconcepts,

$$CC(C_i,\mathcal{H}^{\mathcal{C}}) := \{C_j \in \mathcal{C} \mid \mathcal{H}^{\mathcal{C}}(C_i,C_j) \vee \mathcal{H}^{\mathcal{C}}(C_j,C_i) \vee C_i = C_j\}$$

is used to measure semantic cotopy (SC) between a pair of concepts

$$SC(C_1,\mathcal{H}^{\mathcal{C}}{}_1,C_2,\mathcal{H}^{\mathcal{C}}{}_2) := \frac{|\ CC(C_1,\mathcal{H}^{\mathcal{C}}{}_1)\ \cap\ CC(C_2,\mathcal{H}^{\mathcal{C}}{}_2)\ |}{|\ CC(C_1,\mathcal{H}^{\mathcal{C}}{}_1)\ \cup\ CC(C_2,\mathcal{H}^{\mathcal{C}}{}_2)\ |}.$$

The average semantic cotopy across all pairs of concepts is then used as the basis for an aggregated measure of taxonomic overlap (TO). A similar approach is taken for comparing non-taxonomic relations. Interestingly, the lexical comparisons turned out to be the most useful of these three levels of measures when comparing ontologies. There are, of course, many arbitrary design choices taken in choosing the above measures and different choices may have produced different results.

However, it has been recognised for some time now that any approach to ontology matching that ignores instance data is fundamentally flawed: partly because concept labels are not always human-readable (e.g. they may be numeric identifiers) or use different natural languages (e.g. English and Chinese or English and US English), but mainly because there is insufficient information - even for a human expert - to decide on the best mapping. Hence it is now common practice in the various matching communities to exploit labelled instance data when arriving at a mapping. Shallow matching is still used and researched, but only as a component in matching systems that combine decisions from a range of matchers [DMD+03, BMPQ04, ZL04a]. Not surprisingly, ontology matchers that incorporate instance data far outstrip the accuracy of simple shallow matchers. By taking ontology instance data into consideration, not only is additional information available to the matcher but also ontology matching is able to draw on established machine learning and statistical techniques used in the database deduplication and record linkage fields. In many ways, the ontology mapping problem is closely related to the record linkage problem in the presence of structured background knowledge, although the emphasis is different in each case.

Most ontology matching is still done manually by knowledge engineers or software developers and has been reported as accounting for 60-80% of the resources in a data sharing project [DNH04]. Unfortunately, because of the subjective nature of ontologies and ontology mapping, evaluating the effectiveness of an ontology matcher is difficult and no less subjective. The most popular quantitative method is to compare the output mapping of an ontology matcher against a, so called, *gold standard* reference mapping produced by one or more human experts in the domain.

Interestingly, the machine learning techniques used to identify mappings between ontologies, intuitively at least, are recreating from instance data much of the very same knowledge that is expressed by the ontologies themselves. This begs the question as to whether, in our setting, matching can be just as effectively performed without human-engineered ontologies. Evidence that just such an approach is feasible, at

least at the lower end of ontological complexity, comes from the XTRACT system [GGR + 03]. XTRACT is a deployed system for inferring a Document Type Descriptor (DTD) schema for a database of XML documents. DTDs are not mandatory for XML documents and it is frequently the case that no DTD exists for a given document collection. Hence being able to create a DTD automatically provides potentially useful structural relationship information that may be used by machine learners. The inference algorithms in XTRACT use a three-step process: (1) finding patterns in the input sequences and replacing them with regular expressions to generate 'general' candidate DTDs, (2) factoring candidate DTDs using adaptations of algorithms from the logic optimization literature, and (3) applying the Minimum Description Length (MDL) principle to find the best DTD among the candidates. The system performed well, identifying DTDs which were fairly complex and contained factors, metacharacters and nested regular expression terms. Notably, the resultant DTDs compared well with human created ones for the same data, the learnt ones often being equally as readable as the manually created ones. The readability (a subjective criterion) is important here more to increase human confidence in the result when, in fact, computationally, less readable solutions would be equivalent.

Historically, the digital library community [Int05] has been at the forefront of developments in the creation and adoption of basic ontological forms including taxonomies and thesauri. Many of these pre-date the digital era and are migrations to the computer of older paper-based schemes. Ontologies have been an active research field for decades within the knowledge representation community [Sow99] where much effort since the 1980s has been invested in producing large-scale ontologies. Some of the more widely cited examples are:

- CYC: over a million human-defined rules and ground facts as a knowledge-base engineered to encode human-inspired *common-sense* grouped into multiple locally consistent domains.
- SUMO: Suggested Upper Merged Ontology, a candidate IEEE standard, is an *upper level* ontology that defines entities that do not belong to a specific domain and is designed as a "semantic ground truth" to which domain-specific ontologies can be mapped.
- WordNet: is a lexicon for the (US) English language that groups words and definitions into semantic groups to combine the functions of a dictionary and thesaurus suitable for automated text processing.

One of the most consistent findings in the literature on ontology engineering is that ontologies are notoriously complex to develop, maintain, use and match. Now, interest in ontologies and the challenges of ontology engineering have spread to a whole new audience through the recent adoption of the Web Ontology Language (OWL) [MvH12] as a core component of the Semantic Web initiative [KM01]. Ontology matching had previously been predominantly a specialist activity in the knowledge representation community but, arguably because of the Semantic Web, is now becoming an important practical problem in modern Web application development. OWL, based around the earlier DAML+OIL language [CvHH + 01], is now the W3C Recommendation for ontology representation, with many earlier ontologies having

already been mapped into OWL - including SUMO and WordNet. OWL is an RDF vocabulary that describes relations between classes, cardinality, equality, richer typing, characteristics of properties, and enumerated classes. For full details, see the W3C OWL Web Ontology Language Specification [MvH12].

In summary, schema and ontology matching are concerned with matching background knowledge in the presence of instance data whereas *matching* (i.e. $\approx$) in our context is concerned with matching instances, possibly in the presence of background knowledge. The former is usually considered a harder problem than the latter, largely because it relies on subjective human-engineered knowledge and consistent manual labelling of data to create profiles. The kinds of the background knowledge expressed in an ontology, particularly taxonomies and thesauri, has been shown to be useful in resolving co-references. Learning or otherwise extracting human-readable schema from data is possible and may bypass the need to use human-engineered versions.

6.2.2 Dataspaces

Dataspaces have already been described in "Section 1.2.3 – Dataspaces" (p.18) of our introductory chapter where we described them in terms of their pragmatic and economical approach to data integration that defers semantic integration, such as that discussed for schema and ontology matching from the previous section, until it is required [FHM05, HFM06, FHM08]. Dataspaces take a co-existence approach to data sources. For example, data integration utilising a data warehouse will import data from its original data sources and align it with a unified schema to provide querying of the unified dataaset. By contrast, a dataspace leaves the data in its original data sources and under control of their respective systems; data sources in a dataspace are accessed using a DataSpace Support Platform (DSSP) which layers as an additional architectural tier over these data sources. It is only when a requirement emerges for data integration between data sources in the DSSP that the toolbox of profiling and matching techniques described over the previous sections is brought to bear on the problem.

6.2.3 Discussion of Profiling and Matching Approaches

Although founded in traditional databases, taken at the more abstracted level of profiling and matching arbitrary data sources, the research problems of database reduplication and record linkage have much in common with our own work. Indeed, they directly motivate our work in "Chapter 5 – Querying and Merging Heterogeneous Data" where we investigate the idea of upgrading the relational model to work with heterogenous structured data. However, there is an important difference between the optimal linkage of data sources and our own focus on matching individual descriptions within data sources. The difference is most easily stated in terms of the submission sifting problem: we are not concerned with the optimal assignment of papers to reviewers; we are only concerned with making recommendations to each reviewer. This point is elaborated on in the next section where we discuss the scope of our research.

Couched in our notation, information retrieval is a profiling and matching problem that shares the same form as record linkage and hence the above comments apply

there too. The vector space model from information retrieval is our chosen "tried and trusted" algorithm for profiling and matching of textual content in our motivating use cases. Using this approach avoids the need to involve manually created profiles while at the same time provides a matching algorithm that is known to perform adequately, without the need for training or tuning, over a wide range of textual data.

In our discussion of record linkage we highlighted the availability of similarity-based methods that had been upgraded for structured data. For our work we chose to follow the approach taken in [GLF04], which defines a default kernel for basic terms in a higher-order logic. Both the logic and the kernel are described in "Chapter 2 – Background". The higher-order logic representation forms the basis of our knowledge representation in later chapters and the kernel is used to demonstrate the feasibility of approximate match queries on structured data as part of "Chapter 5 – Querying and Merging Heterogeneous Data". The choice of the default kernel for basic terms is not a specific requirement for the approximate match queries; any distance for basic terms would be suitable. Prior work on distances for logical terms includes distances between Herbrand interpretations [NC97] and between first-order terms (including structures and lists) [Seb97, BHW98, KW00]. None of these directly apply to basic terms and while it may be possible to apply distances on first-order terms to our first-order representation of basic terms, the semantics of basic abstractions would be lost as a result. Most closely related to our work, are various similarity-based methods that have been upgraded to handle structured data [GLF04, BG05, WKKH05]. Contrasting approaches apply probabilistic models to take account of dependencies between resolution decisions [PD04, BG06]. Most recently, a family of pseudo-distances over the set of objects in a knowledge base has been introduced although not specifically for basic terms [dFE07].

We suspect that ontological data in the research intelligence domain will gain importance over time and in future will become more relevant to this Thesis. However, at the time of writing (2014), there are multiple competing vocabularies that themselves require mappings between them before they can be used here. Even the seemingly trivial concept of assigning a unique identifier to researchers remains work-in-progress in the Research Data Management community, with multiple competing conventions. So, while there are a number of useful ideas from schema and ontology matching it is not a good model for profiling and matching in our context. One particularly important lesson from this review that informed the design of our own frameworks is the need to provide human-understandable justification for a matching decision in order to achieve user acceptance.

Avoiding the need for semantic integration of data where not absolutely necessary is a goal that our work on the higher-order relational algebra in Chapter 5 shares with dataspaces. There, as in our own work, the preference is to use automated profile generation and matching in preference to manually created mappings via schema and ontology alignment. We anticipate that dataspaces would be a natural application area for an implementation of approximate relational joins (Section 5.4, p.162).

6.3 Comparisons with Other Frameworks

In this section we compare both our theoretical and software frameworks with similar previous work, respectively in "6.3.1 Comparison with Other Formalisms", immediate below, and "6.3.2 Comparison with Other Software Frameworks".

6.3.1 Comparison with Other Formalisms

In "Section 1.2.1 – Dataflows" (p.6) we surveyed similar efforts to formalise dataflow and workflow. Below we compare the most relevant of these against our formalism introduced in "Section 4 – Higher-Order Dataflows" (p.99) using basic terms in a higher-order logic introduced in Section 2.3.3 (p.52) of our background chapter.

One earlier formal framework for describing dataflows [BNTW95] that has some commonalities with our own framework uses nested relational calculus restriction of structural recursion to ensure that programs involving structured data in a database query language are well-defined. Our higher-order dataflow framework also uses structural recursion, i.e. decomposing a structure as we described in Section 2.2.4 (p.46). The kernel for basic terms, as well as some the definitions and proofs, in Chapter 4 use induction on the structure of basic terms, which is structural recursion by another name. However, [BNTW95] strongly advocates their own approach as alternative to higher-order logic or other extensions to first-order logic on the grounds that these formalisms cannot support multisets. Lloyd's logic, as used in this Thesis, demonstrates that this is not the case by defining multisets extensionally as lambda expressions of the form, $\lambda x.$ if $x = t_1$ then s_1 else $\ldots$ if $x = t_n$ then s_n else s_0, where $t_1, s_1, \ldots, s_n$ are basic terms and s_0 is a default term. Details of these basic abstractions and their capacity for representing multisets appears in Section 2.3.3 (p.52).

Another formalisation of dataflows, this time modelling a deployed and widely used workflow engine, describes Taverna [HWS+06] in terms of a computational lambda calculus with monads embedded into its type system [Mog91, Mog89], as described in Section 1.2.1 (p.6) of our introductory chapter. Their formalism is used to define the sytax and semantics of Taverna, which at the same time implicitly defines of Taverna's underlying dataflow model [TMG+07]; our own dataflow formalism does not currenty extend to the definition of the formal syntax and semantics of JSONMatch and only defines the underlying higher-order dataflow model.

However, at the lower level of the monads used to augment Taverna's computational lambda calculus there is a strong relationship between the Kleisli triple, $(M, unit, \star)$ of a monad [Wad90] and the higher-order template parameters of the transformations in our model. The template parameters can be used in a similar way to monads so that, for example, side-effects to otherwise pure functions can be modelled through embedded functions relation, item and lambda template terms. This echoes the way that a monad can model similar side-effects through functions embedded within the elements of the Kleisli triple. One difference between the two approaches though is that monads in Taverna conceal the complexities of side-effects from the user whereas our model forces the user to specify them at runtime in the form the higher-order template parameters. There is a trade-off then between the two approaches whereby Taverna cleanly separates lower level mechanics of dataflow from the user but at the same time locks down those behaviours encapsulated in monads,

and our pushes those same mechanics out to the user, giving them lower-level control of behaviours in exchange for having to always specify those behaviours in higher-order parameters at runtime.

A third approach which we touch upon briefly is a Haskell description of Ptolomey II workflows [LA03]. Some of the advantages of this approach to modelling, although as the authors stress, not implementing the underlying dataflow are the same claimed in our own higher-order formulations of frameworks in Chapters 4 and 5. One of the main ones is that aa a higher-order function language, Haskell enables declarative definition of dataflow by exploiting strong typing and a fairly intuitive syntax for modelling data; the same is true of our own formalism with regard to modelling types although, perhaps for programmers who struggle with mathematical notation, the readability of our formalism could be enhanced by expressing it using Haskell. In fact, this is exactly what we have done in Chapter 5 where we expressed the type structure for `publications` using Haskell syntax is Example 5.2.14 (156).

In rounding off this comparison of our work to other formalisation efforts, we turn finally to the theoretical framework introduced in Chapter 5, where we used the same higher-order logic to define the higher-order relational algebra described in "Section 5.3.4 – Basic Term Relational Algebra" (p.161). The same formalism [BNTW95] discussed in the context of dataflow formalisms above, also uses types and structural recursion on sets to query complex objects – an approach which has features in common with our higher-order relational algebra framework from Chapter 5. In our algebra we use the same higher-order logic used to define our dataflow framework and so all the previous comments about structural recursions and induction of the structure of terms hold here too. However, it has been shown previously that Codd's relational algebra and relational calculus are equivalent [Cod79], so perhaps the same might be true between (some parts of) our higher-order relational algebra and the nested relational calculus.

6.3.2 Comparison with Other Software Frameworks

The first comparison we make in this section is between our submission sifting web services framework, SubSift, and information retrieval systems. We introduced this topic in Section 2.2.3 (p.44) of our background chapter in Section 3.1 (p.68) where we discussed the rationale of our choice of technology for the implementation of SubSift Services. Here we complete our discussion in light of experience with SubSift.

SubSift's functionality is achieved through the application of techniques from information retrieval that are more normally associated with *full text search* rather than profiling or matching entire collections of documents [SWY75a]. In our background chapter, "Section 2.3.1 – Vector Space Model", we described the tf-idf term weighting and cosine similarity methods involved. Established full text search tools such as Apache Lucene[4], recent versions of PostgreSQL[5] and Oracle Ultra Search[6] all support text matching on large-scale document collections. Although these systems were all designed to compare a single text query against a document collection, there

[4]Lucene – `http://lucene.apache.org`, visited April 2014.

[5]PostgreSQL – `http://www.postgresql.org`, visited April 2014.

[6]Ultra Search – `http://docs.oracle.com/cd/B13789_01/ultra.101/b10731/over.htm`, visited April 2014.

is no reason why a full Cartesian product of one document collection against another document collection cannot be performed to return pairwise document similarity in the same way that SubSift does. Indeed, a paper duplicate detection system using Lucene [DYX10] has been developed to do just that. As touched on in "Section 3.1 – Choice of Technology", there are a number of advantages in implementing SubSift as a bespoke framework instead of using a full text search tools, including the following.

- Detailed metadata about factors contributing to term importance and similarity are available throughout.
- Code to collect this metadata can be more easily integrated into the core algorithm without having to maintain patches or plug-ins for third-party tools.
- A consistent API is available for both the processing and the supporting data management, reporting and publication functionality.

More recently, Big Data approaches to the full-text search problem have been based on indexing text using MapReduce tools [DG08]. However, such tools still do not negate the above advantages of SubSift's bespoke framework.

In "Section 1.2.2 – Workflows" (p.10) in the introductory chapter of this Thesis we surveyed some popular contemporary workflow systems. Although the usage of the terms *dataflow* and *workflow* is far from consistent in the literature, in our work we assume that dataflow frameworks underpin and drive workflow systems (i.e. define their behaviour), as described in Section 1.2.1 (p.6). Implementation of workflow systems, such as Kepler and Taverna [ABJ + 04, HWS + 06], typically incorporate job scheduling and management capabilities as well as enacting dataflows underpinning workflows; our SubSift and JSONMatch systems only feature a minimal workflow language (described in the next chapter). JSONMatch was designed from the outset as a dataflow system controlled by HTTP methods and user-specified dataflows defined during enactment; by contrast, dataflow in SubSift is partly hard-coded into the framework itself. However, in principle, it is possible to describe dataflows from either framework in a workflow language and to share them through workflow repositories like myExperiment [DRGS09].

We next compare JSONMatch to the Taverna system [HWS + 06, SHMG10] as a representative example of a state-of-the-art workflow engine supported by a well-defined dataflow model. Firstly, there are some obvious equivalences between the two systems, for example the product generator in JSONMatch corresponds to the cross product operator in Taverna. Restrictive cross products, as would be required to implement our higher-order relational joins as defined in Chapter 5, are possible in both systems by applying a restriction predicate on the product to achieve what is, in effect, a restrictive cross product. Taverna is a far more mature implementation than the JSONMatch and so the latter lacks many of Taverna's commands and features. While some of Taverna's features could be added to JSONMatch by developing extra libraries of embedded functions, as described in Appendix B, or emulated using sequences of transformations, there are certain features that have no counterpart. One such feature is Taverna's *pipelining* where a subsequent component in a workflow will receive outputs from a previous component one at a time rather than waiting for the component to compute all items before emitting them. This behaviour, potentially, enables the processing of streaming data or promotes concurrency where the

infrastructure supports it. JSONMatch does not proceed to the next transformation in a workflow until all items in the current transformation have been stored in the output relation. Ironically, JSONMatch already emits items one at a time and these items become readable immediately by other transformations, so there is no reason why the same feature could not be added.

An interesting parallel exists between embedded functions in JSONMatch and the user-definable *local worker* components in Taverna[7]. In Taverna these scripts are used to manipulate inputs so as to construct parameters in the required format for inputs to components in the workflow. Embedded functions in JSONMatch can be used to achieve much the same behaviour. However, architecturally there is a substantial difference between the two approaches: the transformations and relations used by the dataflow model underpinning JSONMatch requires that relations will be stored between transformations, i.e. the output of every transformation is a basic relation. This behaviour ensures that the model is, in principle, embarrassingly parallel. By contrast, Taverna does not make such strict architectural assumptions and so gains more control (internally) over the execution of each step in a workflow, for example, having a choice about whether to write intermediate values of a cross-product to store or to cache them in working memory for better performance. The reward for JSONMatch in adopting the higher-order dataflow model is that pure transformations are parallelisable and, even in their serial form, are scalable to large datasets, as discussed in "Section 4.5 – Comparison of JSONMatch with SubSift" (p.132).

We conclude our comparison in the current section by noting that the SubSift and JSONMatch REST services and the workflows presented in this Thesis are compatible with workflow management systems such as Kepler, Taverna and YAWL [ABJ+04, HWS+06, vdAtH05]. A basic implementation could be achieved for any of these systems using a scripting language to invoke REST services and packaging that control script as a local workflow component; the resultant component could then be invoked by the workflow engine to indirectly invoke the remote services. For workflow systems that have built-in REST invocation features such scripts can be avoided by registering the services directly within components which are then invoked through these components. Compatibility would be further increased by the creation of a WSDL 2.0 description of SubSift's REST API which would add SOAP support to simplify integration into existing workflow tools [LGS07, SGL07].

6.4 Scope

Before concluding this chapter, in this section we first mention a number areas of "unrelated work" that are not the subject of our research questions and so fall outside the scope of our Thesis.

- **Profiling and Matching Algorithms** – The availability of sufficiently useful methods for characterising and comparing textual and other structured data is necessary in order for us to explore frameworks for profiling and matching. Our research objectives do not, however, require that these algorithms be the best in their class. So, apart from offering a range of parameters for their con-

[7] These are implemented as bean-shell scripts in Taverna 2.

figuration in Chapter 3 and an extensible mechanism for providing alternative algorithms in Chapter 4, we do not especially focus on the relative performance of the profiling and matching algorithms used in our work. Instead, we trust to the prior work on these algorithms described earlier in this section. The one exception to this occurs in Chapter 5 where we follow a line of investigation suggesting that the structure of data itself might be useful in the matching process. But even there, our research objective is only to establish a proof of concept using an existing algorithm in this novel setting. We could perhaps have paraphrased this bullet by simply stating that this is not a *machine learning*, *data mining* or *information retrieval* Thesis, although methods from each of these fields are employed in our work.

- **Paper Allocation** – We contrast our submission sifting use case with the broader task of optimising the allocation of papers to reviewers under a set of constraints. Submission sifting is concerned with what is generally referred to in the paper allocation literature as a *language model* of matching, using the similarity between textual data derived from submitted papers and the published works of potential reviewers, possibly augmented by other background knowledge. Paper allocation accepts such similarity data as its input and then finds specific assignments of papers to reviewers under constraints such as "each paper must have at least 3 reviewers" and "each reviewer must not be assigned more than 5 papers". For the KDD'09 paper allocation, Flach and Golénia addressed this task by transforming it into an integer programming problem that was solved with a standard solver [FSG+09]. Earlier work in this area is described in [BL01], whereas [CZB11] describes more recent developments that have subsequently been incorporated into a conference management tool. For the proof of concept frameworks described in this Thesis, the paper allocation decisions were left to human experts in the Programme Committee of the relevant conference or to the editors of the journals.

These topics were not covered elsewhere in this "Related Work" chapter and we have only touched on them briefly here to clarify our research objectives from Chapter 1.

6.5 Summary

In this chapter we surveyed a broad range of related research topics and positioned our theoretical frameworks and software frameworks within these areas. In a more narrowly focused section, we then reviewed the literature that led to our choice of algorithms and approach to profiling and matching, as well as to our rejection of some candidate alternative approaches. In the penultimate section we compared our frameworks to existing frameworks. And in the final section, we briefly mentioned some prior work that falls outside of the scope of this Thesis.

Chapter 7

Discussion

In this chapter we review the degree to which our research objectives have been met and then discuss the limitations of our work. We also bring together a list of the applications of our work that we are aware of to-date.

7.1 Review of Research Objectives

In section "1.3 Research Questions and Objectives" of the introduction to this Thesis we distilled our research questions down to four summary research objectives **O1-4**. In this section we review whether these objectives have been met.

O1. *Decomposition of the submission sifting use case into separately re-usable components of a flexible framework for profiling and matching textual content – capable of implementing our research intelligence use cases.*

In "Chapter 3 – Workflows for Profiling and Matching Textual Content", we defined just such a decomposition, consisting of separate profile and match components assembled into the submission sifting workflow. We then demonstrated its utility through SubSift, our proof of concept implementation, which has since been used to support several major machine learning and data mining conferences (listed later in this chapter). Later in Chapter 3, we used a generic submission sifting workflow to define specific workflows that addressed each of the use cases. We also demonstrated that SubSift implementations of these workflows constituted a proof of concept realisation of this flexible profiling and matching framework. One of the use case demonstrators is currently used as a recommender system by editors of two leading computer science journals (listed later in this chapter).

O2. *Description and implementation of a general purpose framework for profiling and matching heterogeneous data in a web application context.*

In "Chapter 4 – Higher-Order Dataflows", we introduced a higher-order dataflow model capable of analysing a wide variety of unstructured, semi-structured and structured data. We demonstrated using JSONMatch, our proof of concept implementation, that the model is able to implement submission sifting workflows through concise user-supplied higher-order parameters to a dataflow at runtime,

rather than through extensive application-specific code that would otherwise be required. Moreover, the built-in functionality of the JSONMatch framework may be extended to include arbitrary functionality defined through third-party web services. Overall, we suggest that this model provides a powerful workflow-based approach to analysing heterogeneous data in a web application context.

O3. *Identification and explorative investigation of deeper design patterns or wider applications suggested in addressing objectives **O1-2**.*

In "Chapter 5 – Querying and Merging Heterogeneous Data", we explored whether matching of heterogeneous structured data could be expressed as declarative queries along similar lines to SQL queries in the relational model, rather than through the imperative domain-specific language used in our higher-order dataflow model from **O2**. To do this we first upgraded Codd's relational model to a higher-order relational model that is better suited to the representation of structured data. We then demonstrated experimentally that matching implemented as approximate joins on structured data in this model has promise for future work.

O4. *Demonstrate the joint potential of* higher-order computational logic *and* workflow systems *for addressing software engineering problems.*

In "Chapter 4 – Higher-Order Dataflows", we demonstrated that terms from a higher-order logic could be used to define the behaviour of a workflow at runtime rather than hard-coding the behaviour at design-time. In Section 4.5 we highlighted the engineering advantages (and some disadvantages) of this approach in the context of interactive web applications.

Thus this Thesis achieves the stated research objectives and makes a number of contributions that we stated in our introductory chapter and will restate in our concluding chapter. However, having reviewed the degree to which our research objectives, we next move on to discuss the limitations of our work.

7.2 Limitations

Over the following sections we discuss a number of limitations of our work and how they can or have already been addressed.

7.2.1 Use of hard-coded algorithms

In "Chapter 3 – Workflows for Profiling and Matching Textual Content" we used our SubSift web services implementation to demonstrate the flexibility of the generalised submission sifting workflow for profiling and matching general textual content. However, in "Chapter 4 – Higher-Order Dataflows" we remarked that while the use of a generic workflow simplifies re-use of the framework in similar use cases, it results in a large number of API options, so that developers can customise the behaviour of each component, and an inability to make low-level changes to web service components without changing the SubSift software itself. In particular, while our choice of

the vector space model, tf-idf and cosine similarity turned out to be sufficient for our purposes, there are numerous alternatives that have been investigated in the literature and which might perform better in these or other use cases.

Our main achievement in "Chapter 4 – Higher-Order Dataflows" was the development of a novel higher-order dataflow and proof of concept implementation, JSONMatch, to remove these restrictions. To-date JSONMatch has been used to support similar use cases to the SubSift software, e.g. including reviewer recommendation for journals and conferences, but is enabling more flexibility in the design of similarity measures. For example, for the ECML-PKDD'12 conference we produced a paper-reviewer similarity score by taking the weighted combination of three cosine similarity on tf-idf vectors representing text, primary and secondary keywords. This required a bespoke change to the SubSift software itself. Implementing this in JSONMatch was possible using the built-in functions, without having to write additional new code; the weighted combination can be calculated in the item template term parameter. Most of the required embedded functions were themselves adapted from the SubSift code in the first place, so perhaps this is not surprising, but the fact that the modified calculation would be achieved through higher-order parameters does show that there is increased flexibility for users to adjust the behaviour of transformations. JSONMatch has far fewer parameters than SubSift but their expressive and computational power is greater.

Another feature of JSONMatch also enables low-level component behaviour to be changed so that, in addition to the built-in functions associated with the default `"jm:"` namespace, arbitrary additional namespaces may be registered to enable embedding of user-defined functions implemented as calls to external web services as illustrated in Example 7.2.1 below.

Example 7.2.1 (Embedding an External Function) *The JSONMatch higher-order parameter below includes the usual `"item"` template that, once evaluated, will result in an item in the output relation of the transformation – in this case, a sorted list of words with HTML/XML mark-up removed. The `"item"` template contains an embedded function in the `"yahoo_pipe:"` namespace that calls a REST API method on the Yahoo! Pipes website. The Pipe itself is a workflow that queries Microsoft Academic Search, returning publications by a specified author since a specified date.*

```
{
    "init": ["jm:register",
        "yahoo_pipe:ms_academic_search",
        "get",
        "http://pipes.yahoo.com/pipes/pipe.info",
        {"params": {"_id": "c48f16a711095a7b0582208a241457d7"}}
    ],
    "item": ["jm:sort",
        ["jm:to_words",
            ["jm:remove_html",
                ["yahoo_pipe:ms_academic_search", {"params": {
                    "textinput1": "Peter Flach",
                    "numberinput1": "2013"
                }}]
            ]
        ]
```

```
    ]
}
```

Before the "yahoo_pipe:ms_academic_search" external function can be used in a template, it must first be registered in a special initialisation template, "init" that is evaluated before the templates are compiled by JSONMatch. The "init" template produces no output.

7.2.2 Inefficiency of naive HTTP-based workflow enactment

SubSift and, to a lesser extent, JSONMatch REST APIs consist of many fine-grained methods that can be used as individual components in a workflow. The web demonstration workflows in our use cases consist of 5-20 such components. Our earliest implementations enacted these workflows by issuing HTTP requests from the web browser to the REST API on the server – in effect, using the browser as a workflow enactment engine. Each such call required JavaScript code to be written and tested. This was resulting in verbose and time-consuming to develop code, which we alleviated somewhat by implementing a mini-workflow language in JavaScript so that workflows could be specified purely as JSON structures. While this shortened the code and reduced development time, it had no impact on the overall number of HTTP requests issued by the browser. So as a natural evolution of this approach we migrated this mini-language to the server, making it a part of both SubSift and JSONMatch web services, so that a single REST API call can be issued to enact an entire workflow.

This need to support multiple-step REST method calls is one of the clearest lessons learned from developing these web application demonstrators. Incorporating this capability into SubSift itself avoids introducing a dependency on external workflow engines, for even relatively simple workflows such as those in this Thesis. To-date SubSift has been used from C, Java, JavaScript, Perl, Python and Prolog, which all require their own approaches to executing sequences of REST API calls. Avoiding the need to implement these in general-purpose languages has resulted in the more recent applications and tools requiring much less coding to implement. Example 7.2.2 gives a flavour of SubSift's lightweight workflow language for encoding simple sequences of commands.

Example 7.2.2 (SubSift Lightweight Workflow Language) *A SubSift lightweight workflow is specified as a plain text parameter to a single API method call. Each line in the parameter corresponds to a single API method call. The example below deletes and creates/recreates a documents folder called `pc` into which it imports the bookmarks from bookmarks folder `pc`. It then profiles the `pc` documents folder. A match folder is similarly deleted and created/recreated before, finally, a match is performed between the `pc` and `abstracts` profile folders to create the `pc_abs` matches folder.*

```
?, delete, documents/pc
+, post, documents/pc, mode=private
+, post, documents/pc/import/pc
*, head, documents/pc/import/pc
?, delete, profiles/pc
```

```
+, post, profiles/pc/from/pc, "ngrams=1,2"
?, delete, matches/pc_abstracts
+, post, matches/pc_abstracts/profiles/pc/with/abstracts
```

Each command will wait for the previous command to complete - including waiting for all the bookmarks imported into the `pc` folder to be harvested by the web robot (which may take minutes or hours). The action taken in response to the HTTP status returned by each call is determined by the symbol at the start of the line. '`+`' requires an HTTP success code to continue, '`-`' requires a failure code, '`?`' ignores the code, and '`*`' repeats the same command until a failure code is returned. The '`*`' code enables polling to be performed using predicate methods added to SubSift for this precise purpose.

This lightweight workflow representation is executed by an equally lightweight workflow engine. However, the addition of this capability enables the above sequence of REST method calls to be made by a single REST call, greatly simplifying the use of SubSift from general-purpose languages and, for that matter, from specialist workflow languages. Other non-trivial REST APIs might also benefit from a similar approach.

7.2.3 Homepage structural heterogeneity

In "Chapter 4 – Higher-Order Dataflows" we described a number of limitations of the SubSift framework that were uncovered as part of the ExaMiner project and then explained how they could be overcome through the adoption of a higher-order dataflow model. However, one further problem encountered when implementing ExaMiner arose from our flawed assumption that researchers' homepages contained sufficient information to 'mine' using the vector space model. Recall that the aim of ExaMiner was to mine and map the University of Bristol's research landscape based, primarily, on information contained in researcher homepages. While ExaMiner did make use of corporate databases to provide lists of staff, this was mainly a convenience for this pilot project. In fact the ultimate goal was to be able to mine all the necessary data from the public website so that, in future, the same methods could be applied to other universities' websites.

It turned out that the structure of researcher homepages varied enormously from department to department and from person to person – even within this single institution. We had originally envisaged (perhaps naively) that a definitive homepage for every researcher could be found and that that page would contain broadly the same amount of information and, ignoring HTML formatting, the same kinds of information. This turned out not to be true, which was important because the tf-idf weighting scheme assigns low weights to non-discriminating terms far more effectively if those terms occur frequently across the corpus and, also if documents are of broadly similar lengths. The consequence of this variability in researcher homepage contents was that SubSift often generated irrelevant terms as features of the profile which then resulted in irrelevant matches. For example, saying that two researchers were similar because they had the terms, "international" or "conference" in their profiles. Although it would be possible to work around the worst of these obviously irrelevant terms by including them in the stopwords list, with more consistent homepages

the tf-idf scheme would remove these terms without the need for such parameters. Admittedly, the problem was exaggerated because SubSift's in-memory computation meant that matches were performed across a single department rather than across the whole university. Had tf-idf weights been calculated across the whole institution then, we suspect, many of these implicit stopwords would have been assigned low weights. Even so, for reasons we discuss next, this would not have entirely resolved the problem.

Inspection of a range of reseacher homepages revealed that one of the obvious causes of variation was the sparsity of text on homepages where the information about a research is not contained within a single web page but, in fact, is distributed over multiple webpages connected by hyperlinks. As part of the ExaMiner project we modified SubSift's web harvester (crawler) to overcome this problem by harvesting not just the bookmark URL provided for the researcher but also a set of web pages hyperlinked from that homepage. While this overcomes the sparsity problem it also has the potential to draw in a lot of irrelevant web pages too – for example, copyright pages, menu pages, the departmental and institutional homepages. To mitigate against this risk we added a number of optional configuration parameters to the harvester, described in Table 7.1, that determine the rules for crawling pages from a researcher's homepage. These may be specified when issuing a REST method call to queue bookmarks for harvesting, thereby allowing different strategies to be used for different websites – for instance, for departments or universities.

Table 7.1: EXAMINER EXTENSIONS TO SUBSIFT WEB HARVESTER.

Parameter	**Values**	**Default**	**Description**
`breadth`	$\{1,\ldots,200\}$	50	Maximum number of links to be followed per page.
`depth`	$\{0,\ldots,3\}$	0	Maximum depth of crawl from homepage bookmark URL.
`same_domain`	$\{\text{true},\text{false}\}$	true	Whether links must be to the same web domain name.
`same_stem`	$\{\text{true},\text{false}\}$	false	Whether links must be child paths of the homepage.
`threshold`	$[0,1]$	0.7	Minimum *idf* score for links, relative to links in all other pages at same depth (see below).

Of these, the `threshold` parameter requires some further explanation. By crawling links top-down, breadth-first from the homepage URL, we could calculate an idf score for each link at a given depth in the crawl, treating link URLs as terms in the tf-idf weighting scheme from the vector space model (Definition 2.3.2). The idf score, relative to links on all pages appearing at the same crawl depth, indicates how discriminating that link is amongst its peer pages. So, for example, a link to an institutional homepage or a copyright page that occurs on every page will receive an

idf of 0, whereas a link that appears on only one page will receive an idf of 1; other frequencies will result in idf values between 0 and 1. Therefore, by specifying a `threshold` parameter, the user can control the required 'uniqueness' of links to be followed. This allowed ExaMiner's harvesting to be tailored to adapt to the specific structure of departmental websites, overcoming the sparse homepage data problem.

7.2.4 Impact of component substitution

One of the characteristics of using workflows and components for engineering solutions is that the components in workflows may be easily substituted for different ones to create new workflows – achieving new functionality without the need to re-engineer existing functionality and without requiring access to the original developer. This is workflow component substitution and although it is exactly what should be expected as a consequence of using workflows, the fact that it turned out to be readily possible was nice. However, there are some serious restrictions which we discuss after giving examples of component substitution.

Our workflow examples described so far in this Thesis have demonstrated that components can indeed be substituted in this way, but the following example also illustrates that this can readily be done by a third-party. Armed only with the SubSift API documentation, Bailey implemented various Yahoo! Pipes web services[1] to act as more flexible alternatives to SubSift's hard-coded DBLP web service[2]. Three of these pipes (listed in Figure 7.1) extract data from an online bibliography and the fourth extracts data from researcher homepages.

- **SubSift DBLP filter** is a direct replacement for SubSift's own DBLP web service for extracting publication titles (as plain text) from a DBLP web page listing author publications (as HTML). While this Pipe adds no immediate advantage over the original bespoke SubSift DBLP web service, it may do so over time because the style and structure HTML pages tends to change, requiring corresponding updates to the information extraction algorithms. Updates to a Pipe to fix such "screen scraping" algorithms do not require changes to SubSift or its DBLP web services.

- **SubSift DBLP filter with year** addresses a user request from Yang, KDD'10 PC Chair, who wanted to limit the maximum age of publications to be used by SubSift for profiling a researcher. The intuition being that a researcher' interests shift over time and as a consequence, profiles of newer publications will better reflect a researcher's current interests than profiles based on older publications. Figure 7.2 shows the Pipe source (itself a workflow), which extracts publications listed in DBLP going back to a specified year.

- **Microsoft Academic Search** replaces SubSift's DBLP web service with Microsofts automatically extracted bibliographic data. Although Microsoft's data has not been manually curated, it does include paper abstracts as well as titles.

[1] SubSift Pipes – `http://pipes.yahoo.com/pipes/search?r=tag:subsift`, visited April 2014.

[2] SubSift DBLP Search – `http://subsift.ilrt.bris.ac.uk/demo/dblp_search`, visited April 2014.

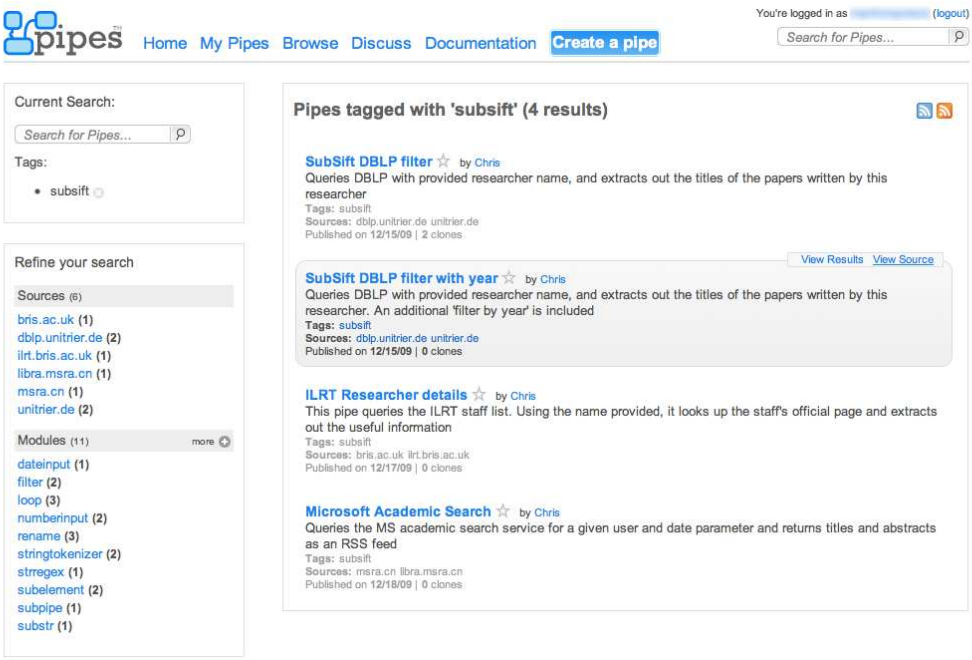


Figure 7.1: Yahoo! Pipes designed as data sources for SubSift (Bailey, 2009).

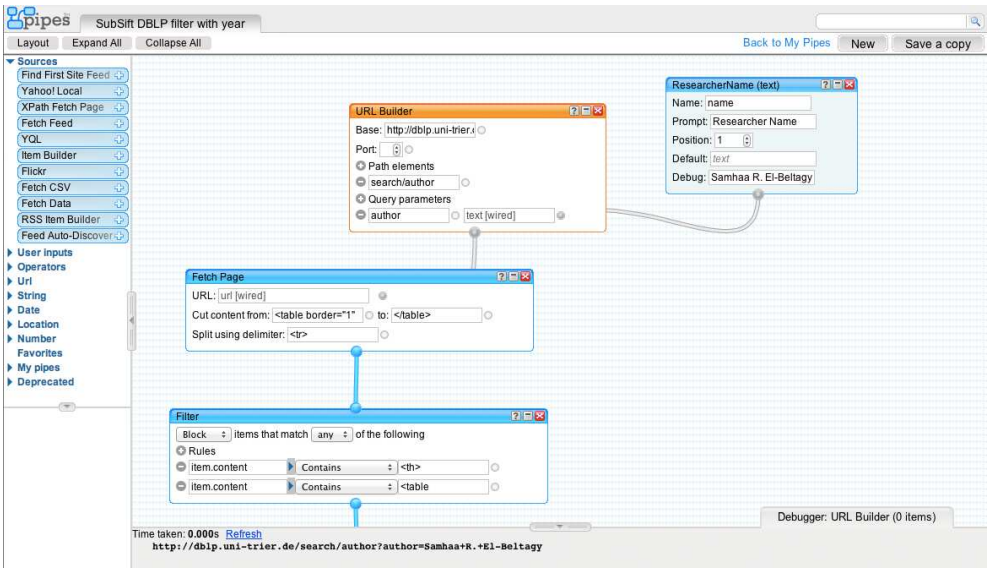


Figure 7.2: Pipe to restrict papers retrieved from DBLP by their age (Bailey, 2009).

- **ILRT Researcher details** extracts researcher data from ILRT staff pages to demonstrate that the clustering work, described later in this chapter, could readily be used to profile arbitrary staff web pages.

There is a downside to the HTTP model adopted when it comes to component substitution. As has already been discussed, the overheads of HTTP do not make its use sensible for local components where high-performance is required. In both

SubSift and JSONMatch, problematic transmission costs associated with invoking components are avoided through the use of a local shared storage layer that all the web services in the API have access to. This way, only metadata describing a method needs to be transmitted to invoke a component, rather than the data itself. So long as the workflow is built of components that have fast access to this shared storage layer then performance is fine. However, as soon as components are substituted for external ones that do not have fast access to the storage layer, the HTTP transmission issues re-emerge. For this reason, the only component substitutions we implemented were those where there was already an implicit HTTP delay anyway, e.g. fetching web pages from remote locations.

7.3 Impact

In this section, in order to convey the impact of the work, we review applications of our models and frameworks to-date. Some of these have been mentioned already in this Thesis and some are introduced here for the first time but we have collected them all together here under the following three headings.

- Academic Peer Review
- Research and Education
- Applied Research Projects

Below we list and discuss the applications that fall into each of these areas in turn. These applications are based on SubSift rather than JSONMatch, which is a much more recent framework. However, where appropriate we add comments discussing how JSONMatch would address particular issues or provide alternative methods of implementation.

7.3.1 Academic Peer Review

Progressive versions of SubSift tools and web applications have been used to support the original "Use Case 1 – Submission Sifting" as part of the wider peer review process for the following conferences.

- **KDD 2009** and **KDD 2010**
  ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.
- **SDM 2010**
  SIAM International Conference on Data Mining.
- **PAKDD 2010**
  Pacific-Asia Conference on Knowledge Discovery and Data Mining.
- **ECML-PKDD 2012**
  European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases.

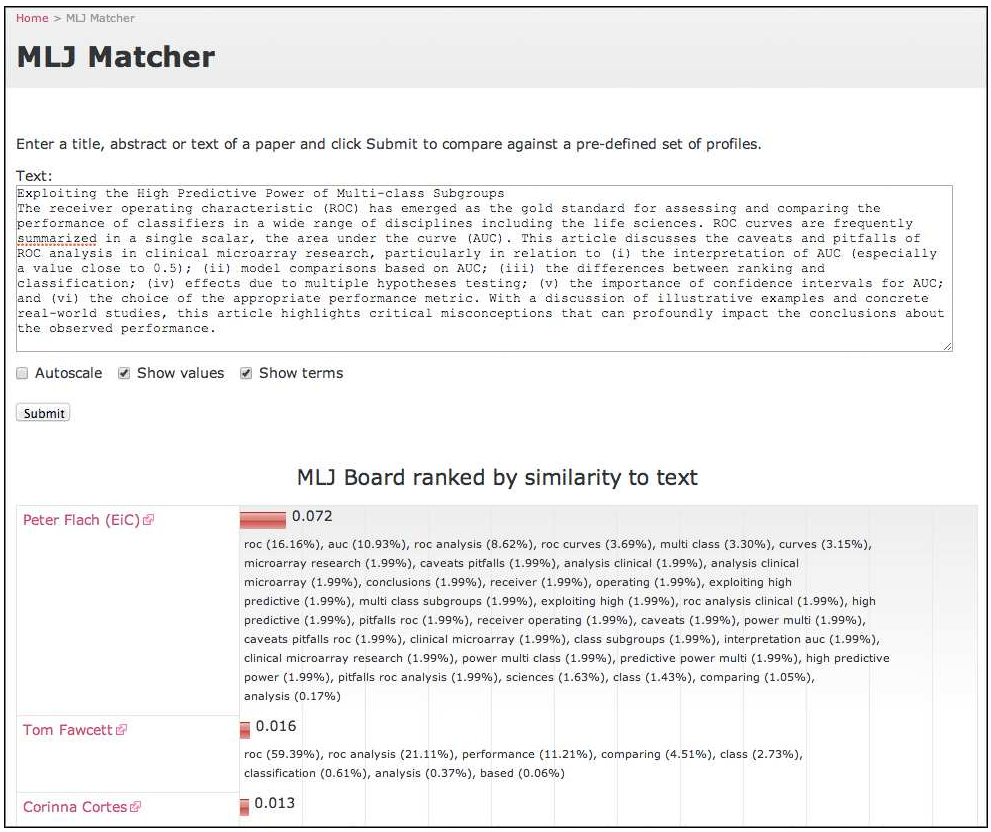


Figure 7.3: Example results from Machine Learning journal Matcher tool.

SubSift workflows implementing variations of "Use Case 2 – Finding an Expert" are being used to support the following two leading international journals.

- **Machine Learning**, Editor-in-Chief: Peter A. Flach, ISSN: 0885-6125 (print), ISSN: 1573-0565 (electronic), since January 2010.
- **Data Mining and Knowledge Discovery**, Editor-in-Chief: Geoffrey I. Webb, ISSN: 1384-5810 (print), ISSN: 1573-756X (electronic), since January 2013.

Each Editor-in-Chief compiled a list of their Editorial Board members' DBLP pages (filtered through the SubSift DBLP web service to extract titles and aggregate multiple DBLP pages relating to the same author). A web application then allows members of each board, usually the Editor-in-Chief and the Area Editors, to paste in the text of paper submissions so that SubSift can match its similarity to the profiles of Board members (Figure 7.3). To support these journals without the requiring ongoing access to a developer, we implemented an additional interface to the "Use Case 2 – Finding an Expert" demonstrator upon which these tools are based (Figure 7.4). The new interface replaces the input to the workflow with an interactive web form that allows the Editor-in-Chief to define, edit and maintain the list of Editorial Board members' URLs themselves.

For the journal expert finding applications, the current SubSift-based implementation of these applications is restricted to relatively small number of potential re-

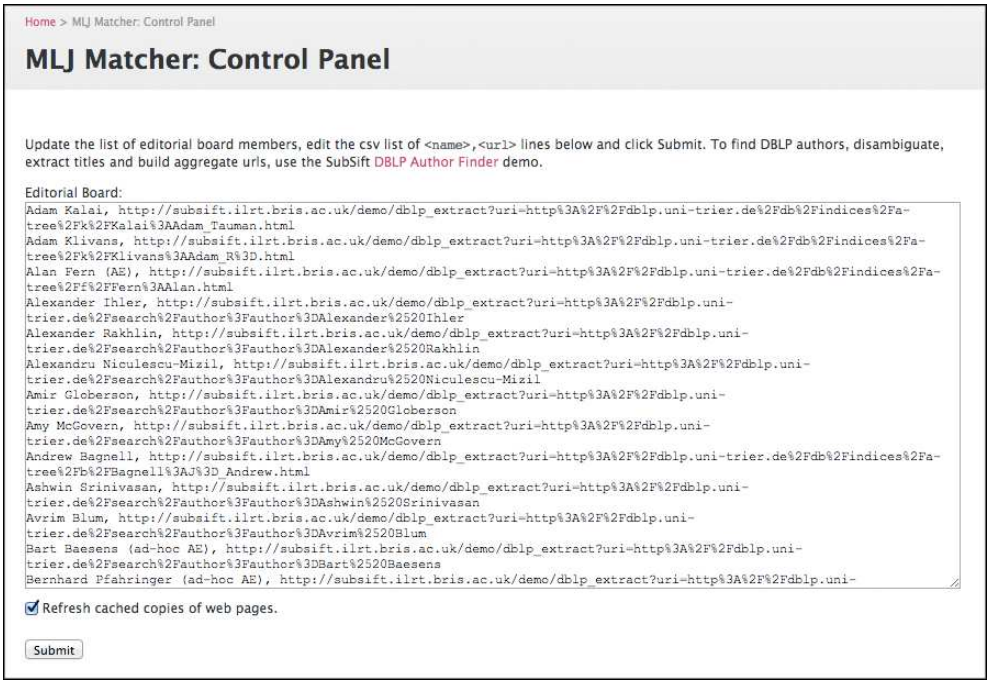


Figure 7.4: Machine Learning journal Editorial Board membership editor.

views because of its in-memory implementation. By constrast, in principle, a JSON-Match implementation would all the numbers to be scaled up to the size of the entire DBLP dataset of over half a million authors provided that relations containing term-document indexes were also generated as well as profiles. Without these additional indexes, the time taken to scan the entire set of authors would be prohbitively slow for an interactive application.

7.3.2 Research and Education

In the introductory chapter of this Thesis we described our own five peer reviewed publications [PF13a, PFS$^+$13, PFS$^+$10d, PFS10b, PF08] and a further seven supporting publications [PF13b, PF13c, CBO$^+$12, PFS10a, FSG$^+$09, Pri04, Pri03] that have resulted from and contributed to this work.

Since 2010, the SubSift web services framework has also contributed to the following five Masters theses [Ham12, Kel12, Abb11, Pen11, Zen10].

- **R. Hambley**. ECMLiPlanner: The intelligent scheduling application. *Masters thesis*, Department of Computer Science, University of Bristol, 2012.
- **T. Kelly**. Timetable scheduling by profiling through SubSift. *Masters thesis*, Department of Computer Science, University of Bristol, 2012.
- **D. Abbas**. Optimal allocation of student projects and markers. *Masters thesis*, Department of Computer Science, University of Bristol, 2011.

- **S. Peng**. Improved matching of potential reviewers to academic papers. *Masters thesis*, Department of Computer Science, University of Bristol, 2011.
- **C. Zeng**. Profiling with SubSift and subgroup discovery. *Masters thesis*, Department of Computer Science, University of Bristol, 2010.

Hambley and Kelly's work was undertaken as a co-ordinated part of the PreSift project first mentioned at the start of Chapter 4 and further described in the next section. Abbas investigated a translation of the submission sifting use case into a different setting – the allocation of student projects and markers. The work used SubSift match data as input to an Answer Set Programming algorithm to find solutions within a set of formally defined constraints. Peng's and Zeng's work also used SubSift match data as input to other machine learning algorithms to enhance results in the submission sifting use case. Peng gathered reviewer feedback on the correctness of SubSift-produced rankings of submitted papers, evaluating improvements in accuracy attainable by incorporating this information into recommendations using different algorithms. Zeng's used CN2-MSD [AF10], a multi-class subgroup discovery algorithm, to find interesting population subgroups in order to predict reviewer preferences for submissions from KDD'09 data.

Taken together, these Masters theses lend support to the idea that our frameworks for profiling and matching do indeed enable the re-use of workflows and components by third-parties. Four of the five Masters theses also successfully used SubSift as part of an interactive web application, the main context of our research, which again lends support to our own Thesis. The fifth used SubSift to pre-process experimental data for offline analysis.

The last case of students using SubSift as data generator has, to some extent, occurred in every student project based on the framework. JSONMatch has a lot to offer students (and users in general) in terms of reorganising and restructuring data into formats required by other tools. This is not a use case we have discussed at any length in this Thesis but our own experience of migrating SubSift data and Weka (.arff) data into JSONMatch has suggested that this type of 'secondary' use of higher-order transformations may be of value.

7.3.3 Applied Research Projects

The proof of concept frameworks that we produced as part of our investigations have been used in two University of Bristol applied research projects – both of which have already been described in earlier chapters.

- **ExaMiner** used the SubSift framework and data from the University's website and corporate databases to produce web applications to mine and map the University's research landscape. (Chapters 3 and 4).
- **PreSift** used SubSift and a prototype of the JSONMatch framework to support academic peer review and personalised timetabling both before and during ECML-PKDD'12 (Chapter 4).

ExaMiner and the peer review elements of PreSift are variations of previously described use cases. However, as well as involving peer review, the PreSift project

for ECML-PKDD'12 also introduced personalised timetabling as an entirely new use case of the SubSift and JSONMatch frameworks. The work was undertaken as part of two separate MSc theses on tight deadlines by Kelly and by Hambley. A supporting PreSift web application to allow delegates to self-register for personalised paper recommendations and personalised timetabling was developed by Louise Millard, Andrew Pickin and the Thesis author. During registration with PreSift, delegates were asked to supply the URL of their DBLP author page or of their web homepage. These URLs were then used by SubSift to profile each delegate so that they could be matched against the accepted conference papers.

Kelly developed *Personalised SubSift Timetable*, a conference scheduler web application that used SubSift match data from a submission sifting workflow to produce a personalised timetable with ratings for each session [Kel12]. Kelly's application also allowed users to produce timetables that rated the best times for a named pair of people to meet at the conference, using each of their individual match results to construct a joint timetable. These users could then look at this timetable to find a time during the paper presentation sessions for which neither user has strong SubSift recommendations.

Hambley developed *ECMLiPlanner*, an iPhone app which also provided recommendations for a personalised schedule based on SubSift match scores [Ham12]. The app was designed for use before and during the conference. Each paper presentation and keynote received a match score that was the result of a comparison of the delegate's profile against the usual titles and abstracts. Also, sessions were assigned scores by computing a combined score from the individual scores of its constituent papers. Figure 7.5 depicts the main screens in Hambley's ECMLiPlanner app.

Hambley's iPhone app and Kelly's web application were both well received at ECML-PKDD'12 and could potentially be re-used for other conferences and events. In fact Kelly's work demonstrated that this was possible by deploying a prototype of the tool at the Movie Comic Media (MCM) Expo, London, May 2012.

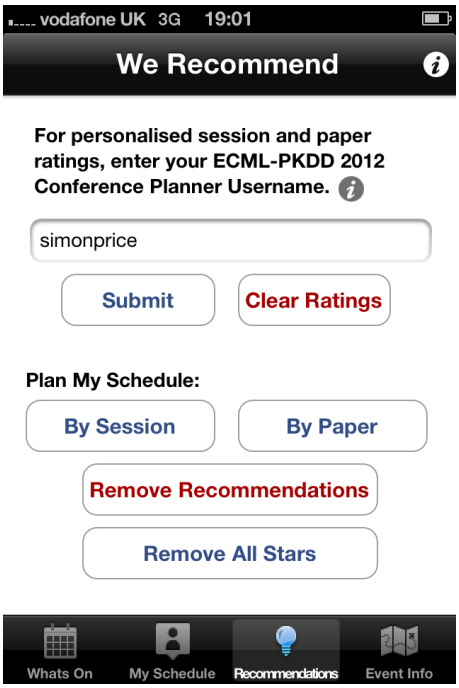
We have already discussed JSONMatch's increased capacity to process larger datasets and this would have been useful here too for both the ExaMiner and PreSift use cases. Perhaps equally as valuable, JSONMatch makes it possible for a user to bring together data from a wider range of formats – both file types and, by virtue of its more sophisticated data access (e.g. extracting sub-parts of data using JSONPath expressions), a variety of structured data.

For the ExaMiner use cases in particular, the most appropriate framework would have been the higher-order relational algebra, providing SQL-like querying of structured data, for which we only have a basic proof of concept implementation at present. However, a robust implementation of the algebra may well be possible by adding a library of additional embedded functions to JSONMatch.

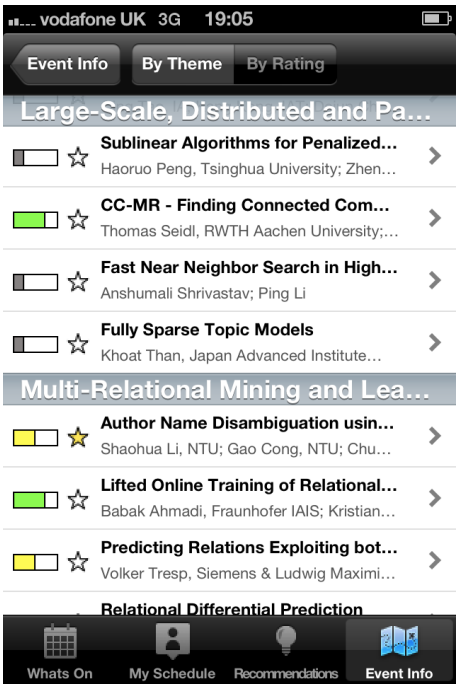
7.4 Summary

In this chapter we concluded that our research objectives have been achieved. However, we also described a number of limitations of the work and how some of these have already of could be overcome. We also brought together a list of the applications of our work in order to convey its impact.

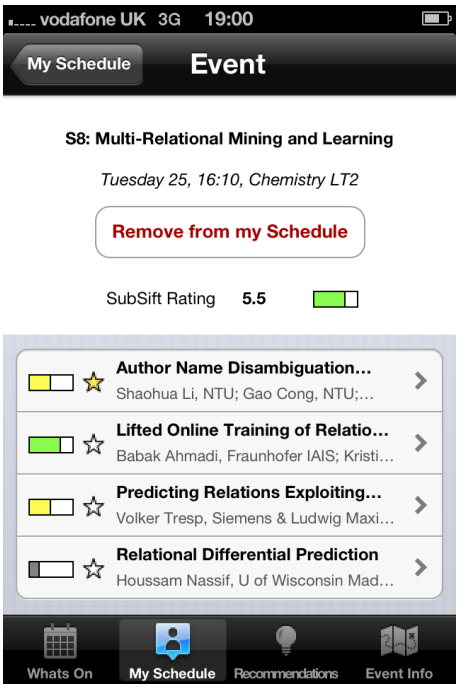
(a) User downloads app from App Store®.



(b) User enters their SubSift username.



(c) SubSift paper *tf-idf* scores as bars.



(d) Scores averaged to rate each session.

Figure 7.5: ECML-PKDD'12 conference planner iPhone app (Hambley 2012).

Chapter 8

Conclusions and Future Work

In this final chapter we draw conclusions from both the planned and serendipitous results of our exploration. We then discuss a number of possible future directions for both the approach and specific implementations featured.

8.1 Conclusions

In this Thesis we explored e-Science and e-Research approaches to the software engineering problem of building *research intelligence* tools that profile and match heterogeneous data about researchers and their organisations. Our exploration began by looking at different solutions to the submission sifting problem and described how an e-Science approach resulted in the SubSift framework for comparing general textual content, which also turned out to be highly applicable to our other use cases. We then addressed a number of shortcomings of purely text-centric solutions by introducing a powerful higher-order representation and dataflow model, generalising the SubSift framework correspondingly to produce the JSONMatch framework. Finally, we speculated that the data comparisons in our use cases may be expressed as queries in a relational algebra defined on terms in a higher-order logic, and reported on encouraging initial results. This exploration resulted in the successful addressing of our research objectives and has contributed to knowledge in the following four ways.

C1. Exploration of workflow approaches to the *submission sifting* problem
Submission sifting is the problem of matching submitted conference or journal papers to potential peer reviewers based on the similarity between the paper's abstract and the reviewer's publications as found in online bibliographic databases. A generic submission sifting workflow and its components are abstracted to define a general framework to enable web services to be assembled into workflows to analyse heterogeneous textual content from web pages and documents. A range of workflows are introduced to investigate the utility of this framework in creating applications to support scientists and their organisations. These applications are, of themselves, not individually novel; the novelty is in their collective definition as workflow compositions of web services. SubSift implementations of these workflows constitute a proof of concept realisation of this general profiling and matching framework. One of the

demonstrator applications is currently used as a recommender system by editors of two leading computer science journals[1]. To the best of our knowledge, despite the potential utility of such a set of services and notwithstanding that the underlying techniques are well established, such an engineering solution is not immediately available elsewhere.

C2. Dataflow model for analysing heterogeneous data

A higher-order dataflow model is introduced. The model ranges over a class of higher-order relations that are sufficiently expressive to represent a wide variety of unstructured, semi-structured and structured data. A proof of concept web services implementation of higher-order dataflows, JSONMatch, is used to demonstrate that the combination of this model and higher-order representation provides a powerful framework for analysing heterogeneous data in a web application context. It is shown that the model has the necessary expressive power to be able to implement submission sifting workflows through concise user-supplied higher-order parameters to a dataflow at runtime rather than through extensive application-specific code that would otherwise be required. It was shown that pure transformations in the model satisfy the conditions of an *embarrassingly parallel* function and hence, in principle at least, are highly parallelisable and highly scalable.

C3. Formalism for querying heterogeneous structured data

The relational model and its associated relational algebra provide a formalism for querying relational data in SQL databases. Hitherto there has been no closely corresponding model and algebra for querying structured data originating from heterogeneous sources. A higher-order relational model is introduced, lifting the traditional relational representation to the basic terms in a higher-order logic that is better suited to the representation of structured data. A relational algebra for basic terms is defined as a single coherent formalism for querying heterogeneous structured data. It is shown that the traditional relational algebra is a special case of the introduced algebra. An extension incorporates approximate joins on complex structured data and is demonstrated to be both feasible have promise for future work.

C4. Bridge between two separate fields of research

Our work brings together the otherwise disparate research traditions of higher-order computational logic and workflow systems; as such, this Thesis constitutes a bridge between these communities and serves to demonstrate their joint potential for addressing problems in research intelligence.

Also, the core content of this Thesis has been peer-reviewed and published in two journal papers and three conference papers and is supported by seven supplementary publications, five of which were peer-reviewed.

[1] Reported by personal communication by Flach (January 2010), Editor-in-Chief, *Machine Learning*, and Webb (January 2013), Editor-in-Chief, *Data Mining and Knowledge Discovery*.

8.2 Future Work

We next present a range of possible future research directions for this work, organised under the following headings.

- Enhancements to Submission Sifting Workflows
- Scaling-up to *Big Volume* Big Data
- Higher-Order Relational Database
- Social Network Analysis
- Research Objects

The chapter, and thesis, then concludes with a brief summary of this future work.

8.2.1 Enhancements to Submission Sifting Workflows

Over the past few years we have responded to suggestions for enhancement from users of the SubSift framework by implementing new functionality or otherwise demonstrating how the same effect could be achieved within the e-Science and e-Research approach without requiring changes to SubSift. However, some of the user suggestions fall outside the scope of our research objectives and questions. Similarly, as part of the overall exploratory work of this Thesis we have identified a number of other potential enhancements that also fall outside the scope of our work. We list the ones we consider most important below.

Integration with a conference management system (CMS) – This is one of the most frequent requests we have received. Integration of the existing submission sifting workflow could be achieved by creating new CMS-specific workflow components to translate between SubSift's input/output data formats and those of the target CMS. Retaining the e-Science approach, communication between the CMS and SubSift could be implemented as web services. Alternatively, the open source SubSift software could be hard-coded into the target CMS. A hybrid solution is also possible whereby a SubSift server is run locally on the same network as the CMS, thereby avoiding sending data over the Internet or external dependencies on the University of Bristol hosted SubSift web service.

Similarity versus importance – A suggestion from PC Chair, Bart Goethals proposed that SubSift reports include a second column showing the rank of that reviewer's similarity to a given paper with respect to that of other reviewers for the same paper. Allowing PC Chairs and reviewers to sort papers on this new attribute would help to inform their decision making.

Finding reviewers for journal submissions – We implemented this expert finding use case at an institutional scale in the ExaMiner project but, intuitively, profiling and matching against the whole of a large-scale online bibliography like DBLP would draw on a much wider pool of potential experts. Scaling SubSift up from hundreds to the thousands of reviewers is possible in our higher-order

dataflow model and, although not tested, in principle using our JSONMatch proof of concept framework. However, JSONMatch's serial execution would be impractically slow for such large-scale datasets and a different implementation of our model would be required. This is scaling-up to Big Volume Big Data is discussed later in this section.

Generating keywords – Conference PC Chair, Qiang Yang asked whether profiles could generate keywords based on an ontology provided by the PC Chair. This is already possible in SubSift to some extent and was demonstrated in our workflow for "Use Case 4 – Profiling Reading Lists" where the set of terms was restricted to come from the ACM Computing Classification System (CCS). However, this lexicon approach is the simplest form of ontology and, as discussed in Chapter 3, richer ontology structures and techniques are available that might be exploited to avoid the need to have an exact literal string match between the term in the text being profiled and the ontological concept.

Receiving such unsolicited qualitative feedback and suggestions has been one of the most encouraging aspects of the submission sifting aspects of our work. It also suggests that, while outside of the research objectives of this Thesis, further work in this area is something that would be of interest to these research communities and, given the widespread practice of academic peer review, possibly others too.

8.2.2 Scaling-up to *Big Volume* Big Data

In "Chapter 4 – Higher-Order Dataflows" we introduced a higher-order dataflow model that supports highly parallelisable design patterns and possesses useful properties for analysing heterogeneous data serially over extended time periods without requiring traditional Big Data computing facilities. JSONMatch is a proof of concept implementation of this model that is capable of processing much larger datasets than SubSift, its predecessor which relied on an in-memory algorithm for both profiling and matching. However, use cases for our existing workflows have emerged that require large-scale storage and high-performance parallel computation – for instance, to profile and compare local document collections against all the documents in a sizeable online bibliography such as DBLP, as mentioned in the "Finding reviewers for journal submissions" request from the previous section.

The workflows presented in this Thesis demonstrate practical applications of SubSift and JSONMatch on relatively small, albeit *Big Variety* Big Data (i.e. heterogeneous data) with modest hardware, we anticipate that these workflows could be used largely unchanged in the *Big Volume* Big Data setting; only the internal behaviour of JSONMatch would differ, most likely as a result of re-implementation using the MapReduce [DG08] paradigm in Apache Hadoop – perhaps using higher-level tools such as Apache Pig.

8.2.3 Higher-Order Relational Database

In "Chapter 5 – Querying and Merging Heterogeneous Data" we defined a higher-order relational model and reported on experimental approximate joins using a prototype implementation. The prototype was written in Prolog in order to take advan-

tage of its representational similarity to the higher-order representation in which our algebra was defined. Prolog also provided an ideal computational paradigm for the evaluation of the default kernel for basic terms on arbitrary structured data, which formed the basis of our approximate joins. However, current Prolog implementations do not lend themselves to the development of large-scale database systems capable of working with datasets of comparable sizes to current relational databases, document-based NoSQL databases, or distributed file stores.

One possible future direction would be to implement the higher-order relational model as a virtual layer over an existing relational database. Another would be to implement it within the framework of a Big Data NoSQL database, such as CouchDB which has built-in MapReduce features, or using a MapReduce tool such as Hadoop, in conjunction with a large-scale distributed file system.

Alternatively, another possible future direction would be to utilise the higher-order dataflow model introduced in "Chapter 4 – Higher-Order Dataflows", and implement the higher-order relational algebra as embedded functions within the template terms of transformations. For similar web application settings to those in this Thesis, it might be possible to use the existing JSONMatch framework; for larger-scale (Big Volume) applications it would be necessary to implement the algebra in a scaled-up Big Data framework, such as the scaled-up JSONMatch envisaged in the previous section.

8.2.4 Social Network Analysis

In this section we outline an exploratory investigation of a potential application of SubSift to social network analysis within an organisation. Figure 8.1 shows a fragment of a dendrogram produced in Matlab by clustering a similarity matrix calculated by SubSift, pairwise matching ILRT staff homepages[2]. The parenthesised numbers to the right of the labels represent each staff member's actual ILRT project group membership. Note that the matrix was first converted from cosine similarity to a normalised cosine distance. These distances were then used as input to an agglomerative (bottom-up) hierarchical clustering algorithm [War63]. Figure 8.2 shows pairwise precision and recall for thresholds at each node in the dendrogram, produced in the same way as for our experiments in Chapter 5. While the precision-recall plot suggests no single ideal threshold, some project groups are rediscovered by the clustering but, more interestingly, the clustering also reveals older project groupings and co-authorships prior to organisational restructuring at ILRT. Potential applications might include locating research bid partners across an organisation or identifying informal structures within an organisation.

8.2.5 Research Objects

Research Objects aim to describe and, where possible, archive both data and computational aspects of research, particularly published research [DR13, BCG+12]. Research Objects are not yet a mature and universally adopted concept but they have received considerable interest in the UK from the Research Data Management community, where funders and publishers are now mandating the publication of research

[2] ILRT – `http://www.bristol.ac.uk/ilrt/`, harvested May 2010.

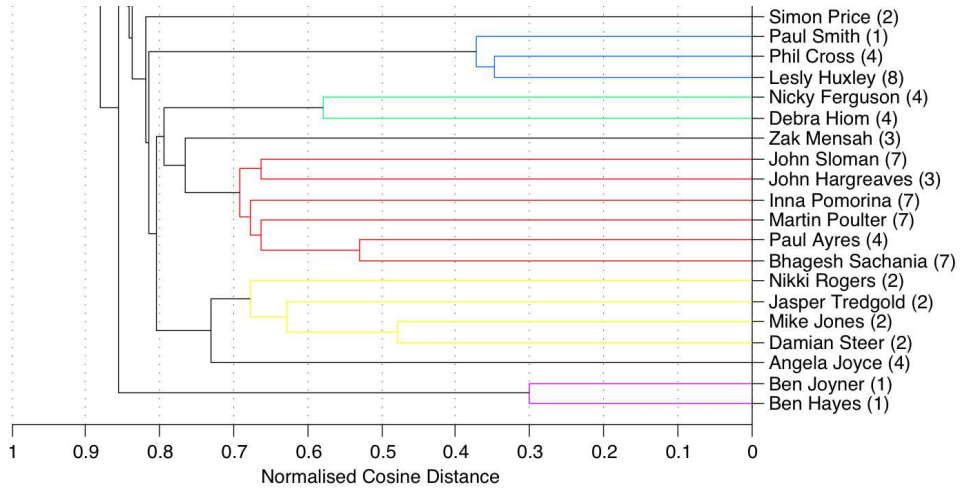


Figure 8.1: Using SubSift similarity data to cluster ILRT staff homepages.

data that supports published research outputs. Although the emphasis in the UK to-date has been on storing and curating research data, the scientific principle of reproducibility seems likely to result in this extending to computational aspects in the future – including the provenance of data, such as where it originated and precisely how it was transformed. Research Objects are intended to capture such information. Therefore, we include Research Objects in this discussion of future directions and consider three aspects that are particularly relevant to this Thesis: research data publication, research data provenance and reproducible computation.

The SubSift and JSONMatch frameworks support the publication of folders and relations respectively to the web. Permissions can be controlled at the level of folders or relations so that, where required, restricted access can be enforced. JSONMatch allows arbitrary structured data to be associated with both relations and the elements within a relation, which might make it possible to add standard DataCite[3] metadata that describes the data contained therein. This could, perhaps, extend to issuing a DataCite Document Object Identifier (DOI) for data.

Importantly for research data publication, our frameworks make data available in multiple formats, including RDF, XML and JSON. When a folder or relation is designated as `public` it is mapped to a publicly accessible URL. However, because the data is simultaneously made available in multiple formats, it can be considered to exist at multiple URLs sharing the same stem but having different file extensions[4] (e.g. `.rdf`, `.xml` and `.json`). Publishing data in this range of formats allows users to download the data in the most convenient format for their work. However the fact that RDF is one of the formats also means that any data published by this method automatically becomes part of the Semantic Web graph and is available in Linked Data format. This aids discoverability of the data but, compared to publishing solely as RDF, does not require that users must access the data as RDF, they can choose whichever format is most convenient for their intended use. Future work could in-

[3]DataCite – `http://www.datacite.org/`, visited April 2014.

[4]Alternatively, HTTP content negotiation may be used to retrieve multiple formats from the same URL stem without the need to specify a file extension.

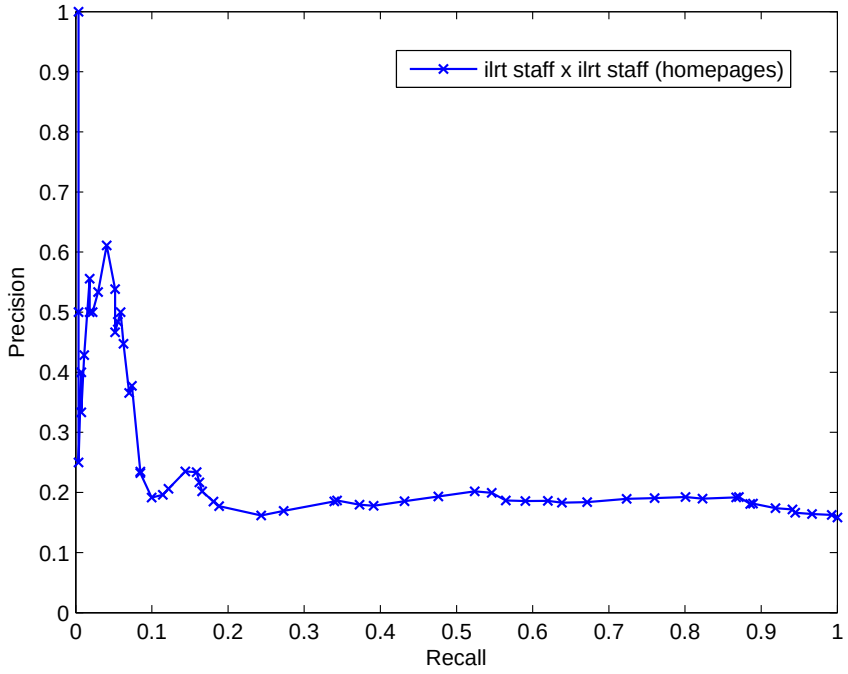


Figure 8.2: Pairwise precision and recall for clustered ILRT staff homepages.

vestigate the potential utility of these features from the perspective of the Semantic Web and Linked Data, for instance investigating whether profile data can be used to link JSONMatch-published data with other datasets in the Linked Data cloud, for example Wordnet or DBPedia (the Linked Data version of Wikipedia).

The web service components of SubSift include methods associated with the parameterised, but hard-coded, algorithms for profiling and matching folders of data items. Workflows composed from these components completely defines their interactions and parameters and consequently have an important role to play in formally recording the provenance of transformations of the workflow's input data. This same recording can be recorded in even more fine-grained detail in JSONMatch, where the transformations are specified within the higher-order parameters to the dataflow. However, an open research problem remains around the archiving of web services' computational behaviour and algorithms in e-Science and e-Research systems. In the case of our frameworks, the availability of the source code makes this possible by archiving the software underpinning the framework. However, where JSONMatch is used to embed third-party web services (e.g. from Yahoo! Pipes), there is currently no established method of archiving this functionality.

Finally, an interesting summary research question that involves investigating all of the ideas discussed in this section is: to what degree can folders and relations in the SubSift or JSONMatch frameworks be treated as Research Objects?

8.2.6 Virtual APIs

One of the lessons from this Thesis is that adding additional flexibility and power to an API comes with a corresponding cost in terms of the amount of work needed for a user to specify a dataflow. The higher-order template parameters to a JSONMatch transformation are non-trivial and can be time-consuming to develop. However, once developed, their re-use (from personal experience) often just involves cut and pasting the same template string into the calling code. Clearly there is a need to create a way to save transformation parameters so that they can be easily re-used, but more importantly so that they can be packaged up to create *Virtual APIs* that could, for example, outwardly create an API that is as easy to use as SubSift's but beneath the surface it is simply a pre-defined set of higher-order templates and workflow sequences. Adding such a layer to JSONMatch would enable different types of user to engage with the system at different levels so that expert users could develop application or domain-specific Virtual APIs that could then be utilised by less experienced users.

8.2.7 Summary of Future Work

In this section we presented a range of possible future research directions for this work. Our own ideas for enhancements to the workflows of our original use cases were described along with those contributed by conference delegates, PC Chairs and other users. Intuitive ideas for scaling-up our frameworks to process Big Volume Big Data, and for scaleable implementation of our higher-order relational model, as well as the notion of a Virtual API, were discussed. We presented an initial investigation into possible applications of profiling and matching in the domain of social network analysis, before concluding with a discussion of broader ranging questions relating to Research Objects and Virtual APIs.

Appendices

Appendix A

A JSONPath Primer

In this appendix we give an introduction to the JSON data format and the JSONPath language. These topics support Chapter 4 – Higher-Order Dataflows.

A.1 JSON

JSON (JavaScript Object Notation) is a data interchange format that is well supported by a wide range of programming languages and applications[1]. JSON has a remarkably simple (and thus far stable) syntax that will be instantly familiar to programmers of C-family languages like C/C++, Java, JavaScript, Perl and Python. Variations or extensions of the JSON syntax are not permitted under the standard [ECM13].

JSON data consists of arrays, denoted `[elem1,elem2,...]`, and objects, denoted `{"key1":elem1, "key2":elem2, ...}` where each `elem` is a string, number, array, object or one of the constants `true`, `false` and `null`. The following is an example of a JSON object representing a publication.

```
{
    "publication": {
        "title":  "Coding guidelines for Prolog",
        "year":  2012,
        "journal": "Theory and Practice of Logic Programming",
        "volume": 12,
        "number": 6,
        "keywords": ["Prolog", "style", "coding standards"],
        "pages": {
            "count": 39,
            "range": [889, 927]
        },
        "authors": [
            {"lastname": "Covington", "initials": ["M", "A"]},
            {"lastname": "Bagnara", "initials": ["R"]},
            {"lastname": "O'Keefe", "initials": ["R", "A"]},
            {"lastname": "Wielemaker", "initials": ["J"]},
            {"lastname": "Price", "initials": ["S"]}
        ]
    }
}
```

[1] JSON – `http://json.org`, visited April 2014.

The outermost object is called `publication` and has properties such as `title`, `year`, etc. The `keywords` property consists of an array of strings. The `pages` property is an object with two properties, the second of which is an array. The `authors` property is an array of objects, each of which is an author's surname and an array of their initials.

JSON is widely used in web services and all modern web browsers have fast built-in JSON parsers and serialisers. The language's simplicity, widespread support and popularity in web services has also led to JSON being used as the default interchange format for document-based NoSQL databases associated with Big Data applications. JSON data is typically transported over HTTP as plain text with the mime type of `application/json`. Although typically more concise that an equivalent XML representation of the same data, different transport serialisations of JSON are used in some document-based systems: for example BSON (Binary JSON) is used as a compressed serialisation to reduce physical storage and bandwidth requirements.

Unlike XML which has a standard XML Schema, JSON currently has no standard JSON Schema. There are currently competing proposals for a standard JSON Schema and, interestingly, most of the candidates are themselves expressed in JSON.

A.2 JSONPath

JSONPath is a language for writing expressions that refer to sub-parts of a JSON data structure [Gös07]. A single JSONPath expression can identify one or more elements in an array, one or more elements in an object, or one or more elements in arbitrarily deeply nested arrays and objects. In both name and design, JSONPath is based on the more widely known XPath language for referring to sub-parts of an XML structure [RCDS14]. However, by design there are a number of important differences between the two path languages:

- JSONPath notation closely corresponds to the native data representations in the C-family of languages. In many cases, JSONPath expressions have syntactically identical expressions in the implementation language.
- JSONPath includes only a minimal subset of the features of XPath. This makes its implementation trivial compared to XPath and also makes it easier for developers to learn and use for common use cases.

JSONPath expressions always begin with a `$` character, which refers to the root of a JSON data structure (i.e. the whole data structure). References to sub-parts of the structure use the familiar *dot* notation from languages such as JavaScript, Java, Python and PHP. Alternatively, the *n*-dimensional array syntax found in all the C-family languages may be used. The two notations may be mixed in the same expression. Examples of both notations are given in Table A.1 for JSONPath expressions referring to sub-parts of the example JSON structure from the previous section.

All the examples in Table A.1 refer to absolute elements within a structure. However the referential power of path expressions comes from their ability to refer to ranges

Table A.1: EXAMPLE JSONPATH ABSOLUTE EXPRESSIONS

JSONPath Expression	JSON Element
`$.publication.title`	`"Coding guidelines for Prolog"`
`$['publication']['title']`	`"Coding guidelines for Prolog"`
`$.publication.year`	`2012`
`$['publication']['year']`	`2012`
`$.publication.keywords.0`	`"Prolog"`
`$['publication']['keywords'][0]`	`"Prolog"`
`$.publication.keywords.1`	`"style"`
`$['publication']['keywords'][1]`	`"style"`
`$.publication.pages`	`{"count":39,"range":[889,927]}`
`$['publication']['pages']`	`{"count":39,"range":[889,927]}`
`$.publication.pages.count`	`39`
`$['publication']['pages']['count']`	`39`
`$.publication.pages.range.0`	`889`
`$['publication']['pages']['range'][0]`	`889`

and subsets of elements at arbitrary nested depths in a JSON structure. To achieve this, JSONPath includes syntax elements for traversing abitrary JSON structures and identifying single or multiple sub-parts of the structure. Table A.2 introduces this syntax by analogy with the syntax of XPath.

In Table A.3 we give example JSONPath expressions applied to the same example JSON `publication` data structure from the previous section. We do not provide examples of JSONPath's `?()` or `()` operators because they are excluded from our work because they are inherently insecure in a web application context and therefore disabled in our JSONMatch application of JSONPath.

JavaScript, C# and PHP implementations of JSONPath are available as open source from the JSONPath project on Google Code[2]. The Perl version used in the JSONMatch implementation from Chapter 4 is available on CPAN[3]. Open source implementations are also available for other popular languages, including C[4], Java[5], Python[6], Ruby[7] and Scala[8]. The interactive JSONPath Expression Tester is a good way to both learn and try out JSONPath without having to write any code[9].

[2]JSONPath project – `https://code.google.com/p/jsonpath/`, visited April 2014.

[3]JSONPath in Perl – `https://metacpan.org/pod/JSON::Path`, visited April 2014.

[4]JSONPath in C – `https://github.com/rogerz/jansson/tree/json_path`, visited April 2014.

[5]JSONPath in Java – `https://url.google.com/p/json-path/`, visited April 2014.

[6]JSONPath in Python – `https://github.com/kennknowles/python-jsonpath-rw`, visited April 2014.

[7]JSONPath in Ruby – `https://github.com/joshbuddy/jsonpath`, visited April 2014.

[8]JSONPath in Scala – `https://github.com/gatling/jsonpath`, visited April 2014.

[9]JSONPath Tester – `http://jsonpath.curiousconcept.com`, visited April 2014

Table A.2: JSONPATH SYNTAX ELEMENTS [Gös07]

XPath	JSONPath	Description
`/`	`$`	Root object/element.
`.`	`@`	Current object/element.
`/`	`. or []`	Child operator.
`..`	`n/a`	Parent operator.
`//`	`..`	Recursive descent. JSONPath borrows this syntax from ECMAScript for XML (E4X)[†]
`*`	`*`	Wildcard. All objects/elements regardless their names.
`@`	`n/a`	Attribute access. JSON structures do not have attributes.
`[]`	`[]`	Subscript operator. XPath uses it to iterate over element collections and for predicates. In JavaScript and JSON it is the native array operator.
`|`	`[,]`	Union operator in XPath results in a combination of node sets. JSONPath allows alternate names or array indices as a set.
`n/a`	`[start:end:step]`	Array slice operator borrowed from ECMAScript (ES4)[†]
`[]`	`?()`	Applies a filter (script) expression.
`n/a`	`()`	Script expression in the native implementation language (e.g. JavaScript).
`()`	`n/a`	Grouping in XPath.

[†] `www.ecma-international.org/publications/standards`, visited April 2014.

Table A.3: EXAMPLE JSONPATH EXPRESSIONS

JSONPath Expression	JSON Elements
`$..title`	`["Coding guidelines for Prolog"]`
`$..count`	`[39]`
`$.publication..count`	`[39]`
`$..range`	`[[889,927]]`
`$..range.*`	`[889,927]`
`$..range[*]`	`[889,927]`
`$..pages[*]`	`[39,[889,927]]`
`$..lastname`	`["Covington","Bagnara",...,"Price"]`
`$..lastname[1,1,4,4]`	`["Bagnara","Bagnara","Price","Price"]`
`$..initials`	`[["M","A"],["R"],["R","A"],["J"],["S"]]`
`$..initials[*]`	`["M","A","R","R","A","J","S"]`
`$..initials[0]`	`["M","R","R","J","S"]`
`$..initials[1]`	`["A","A"]`
`$..initials[1,0]`	`["A","M","R","A","R","J","S"]`
`$..authors[-2:].lastname`	`["Wielemaker","Price"]`
`$..authors[-2:-1]`	`[{"lastname":"Wielemaker","initials":["J"]}]`
`$..authors[-2:-1].*`	`["Wielemaker",["J"]]`

Appendix B

JSONMatch Functions

JSONMatch allows functions to be embedded in the relation template and item template passed to a transformations. These functions all return JSON values (i.e. scalars, arrays or objects). JSONMatch templates are represented as JSON which, by design, is a language without functions. For this reason JSONMatch embedded functions are represented as JSON arrays, where the first element is the name of the function, as a string, and subsequent elements are the arguments of the function, as follows.

```
["namespace:function", arg1,...,argN]
```

JSONMatch has a library of built-in functions that all have names beginning `"jm:"`, designating the JSONMatch namespace. A trivial example is the function `jm:length`, which maps a string to its length, e.g. `["jm:length", "abc"]` will be substituted with `3`. This may be embedded in a template such as the JSON object below.

```
{
  "x": "abc",
  "y": ["jm:length", "abc"]
}
```

After evaluation, this would result in the following JSON object.

```
{
  "x": "abc",
  "y": 3
}
```

To extend the functionality of JSONMatch a mechanism is provided for adding new functions as arbitrary calls to REST API web services. The following section documents this HTTP-based mechanism and is followed by a brief section on alternatives for adding more performant functions locally. JSONMatch continues to evolve and so the definitive documentation is always the online version on the JSONMatch website[1]. The online documentation includes detailed descriptions of each of the

[1] JSONMatch website – `http://jsonmatch.ilrt.bris.ac.uk`, visited April 2014.

built-in functions along with examples of their use. To give a feel for the current capability of the system, the final section in this appendix lists the current (April 2014) range of built-in functions.

B.1 External Extension Functions

The range of functions available for embedding in JSONMatch templates is not restricted to the list of built-in `"jm:"` functions described in the final section of this appendix. Additional functions may be added to a template by registering the url of external RESTful web services and REST API method calls on those services using a special JSONMatch parameter called `init`. These registered url-method pairs may then be embedded in JSONMatch templates using associated `["namespace:function"]` name and used just like the built-in `"jm:function", arg1,...,argN]` functions, but with the requirement that arguments are passed as a JSON object including the key `params`, which has an associated list of arg-value pairs as follows.

```
["namespace:function",
  {"params": {
      arg1: value1,
      arg2: value2,
      ...
      argN: valueN
}}]
```

The registration of an external REST web service method as a new JSONMatch function is illustrated in the following example templates parameter to some transformation. The example registers a (previously unmentioned) SubSift web service method called `search` as a new JSONMatch function called `sift:search`. The registration is achieved by inclusion of one or more `jm:register` function calls in the special `init` template value. No output is produced as a result of JSONMatch evaluating the `init` value and so any valid JSON structure may be used, including none or more `jm:register` function calls. Functions embedded in the `init` value are guaranteed to be evaluated before the more familiar `relation`, `item` and `lambda` templates, which means that the registered functions may be invoked from these templates.

```
{
  "init": ["jm:register",
    "sift:search",
    "get",
    "http://subsift.ilrt.bris.ac.uk/sift/search",
    {"params": {"refresh": 1}}
  ],
  "item": {
    "timestamp": ["jm:timestamp"],
    "description": "Result of searching DBLP for author names",
    "results": ["sift:search", {"params": {
        "names_list": "Peter Flach",
        "refresh": 0
    }}]
```

```
    }
  }
```

The above JSON object also includes an `item` template with two embedded functions. The first invokes a built-in function to insert a timestamp into the resultant item. The second invokes the external SubSift web service `sift:search` to search the DBLP online bibliography for a comma-separated list of author names (just a list of one person in this example). A second parameter, `"refresh":0`, overrides a default of `"refresh":1` defined in the original registration of the external function. This parameter determines whether SubSift's cached copy of the search results should be used or whether a fresh search should be sent to DBLP. The resulting item from a transformation using the above `item` template is shown below (abbreviated for clarity).

```
{
  "timestamp": 1400941364.897874,
  "description": "Result of searching DBLP for author names",
  "results": [
    {
      "name": "Peter Flach",
      "uri": "/db/indices/a-tree/f/Flach:Peter.html"
    },
    {
      "name": "Peter A. Flach",
      "uri": "/db/indices/a-tree/f/Flach:Peter_A=.html"
    },
    {
      "name": "Peter Flachenecker",
      "uri": "/db/indices/a-tree/f/Flachenecker:Peter.html"
    }
  ]
}
```

The `results` key is associated with the value returned by the call to `sift:search` and is an array of matches for each author name passed as an argument. In this example, there are three matches for "Peter Flach" in the DBLP author database.

RESTful web services rely on HTTP, which is well suited to for accessing remote services, particularly where the amount of data to be transferred is not large and the number of calls is relatively small. Bespoke functionality may be usefully incorporated into JSONMatch using this mechanism by writing new web services or by incorporating calls to existing web services, such as those listed in directories like Programmable Web[2]. However, HTTP is inefficient for the inclusion of high-performance functions, particularly where the data to be transferred is large or the number of calls is high. In future, JSONMatch may also support local protocols as a more performant alternative to HTTP, but until then the internal extension mechanism described in the next section offers a pragmatic alternative.

[2]Programmable Web – `http://www.programmableweb.com`, visited April 2014.

B.2 Internal Extension Functions

This section is only relevant if developers have shell or sftp access to the installed folder of a JSONMatch installation, for example when running JSONMatch on their own computer or when they have been granted read-write access to the `functions` folder in an otherwise access-restricted installation as described below. A basic knowledge of Perl programming is also required, although it is not necessary to understand the complexities of the JSONMatch source code apart from the intuitive extension mechanism descibed here.

New JSONMatch functions can be added directly through a purpose-designed scheme that lets programmers add new libraries such that they will be automatically detected by JSONMatch and declared as namespaces with corresponding function collections. This scheme is inspired by the declarative loose coupling approach used in frameworks like Python Django and Ruby Rails, where the system looks for files following a pre-defined naming pattern in a designated folder. In JSONMatch, this folder is called `functions` and is a sub-folder of the application home (installation) folder, i.e.

```
<home_folder>/functions/
```

and any files named `<namespace>_functions.pl` found in there are automatically detected and a corresponding function namespace declared, e.g.

```
<home_folder>/functions/soap_functions.pl
```

would declare a namespace called `soap:`. There is no other registration or linking required. Perl functions in each file that follow a pre-defined naming scheme (i.e. `function_<namespace>_<fn>`) are automatically added to the file's namespace as JSONMatch functions that may be embedded just as with the built-in functions. For example, the following function in a `hello_functions.pl` file,

```
sub function_hello_world {
    my ($message, $times) = @_;
    return $message x $times;  #x is Perl's string repeat operator
}
```

could be invoked from within a JSONMatch template as,

```
["hello:world", "Hello World!", 3]
```

where `hello:` is the namespace, `world` is the JSONMatch function name within that namespace and `"Hello World!", 3` are arguments.

The functions in these libraries do not directly deal with JSON; the input and output of the functions are native Perl data structures. JSONMatch handles the type transformations required to map these values to their JSON equivalents at the REST API level. JSONMatch uses the internal extension mechanism described in this section to implement all the functions in the `jm:` namespace. For a wide range of examples, inspection of the `jm_functions.pl` file in the `functions` folder will reveal the implementation of each of the built-in `jm:` functions listed in the next section.

B.3 Built-in Functions

The built-in JSONMatch functions can be divided into the following categories.

- Meta
- Web
- Time
- Set
- Array
- String
- Text
- Comparison

The tables below list the full range of functions in each of these categories. Further details are available in the online documentation on the JSONMatch website.

Table B.1: JSONMATCH META FUNCTIONS

Function	Description
`apply`	Apply higher-order JSON template to each element of an array.
`restrict`	Select values in an array that satisfy a predicate.
`generate`	As apply but outputs results as new relation items.
`project`	Subpart/s of a value identified by a JSONPath expression.
`types`	Detect types of all subparts of a JSON value.

Table B.2: JSONMATCH WEB FUNCTIONS

Function	Description
`http_get`	Fetches text of web page; or REST call.
`http_post`	Submits web form; or REST call.
`register`	Register external REST functions.
`rpc`	Remote procedure call to registered external REST functions.

Table B.3: JSONMATCH TIME FUNCTIONS

Function	Description
`time`	Floating seconds since Epoch (using millisecond accuracy).
`interval`	Floating seconds between two times.
`timestamp`	Time when the current transformation started.

Table B.4: JSONMATCH SET FUNCTIONS

Function	Description
`set`	Constructs a key-boolean set from an array of keys.
`set_cardinality`	Cardinality of N key-boolean sets as the number of unique keys.
`set_union`	Union of N key-boolean sets as a key-boolean set.
`multiset`	Constructs a key-multiplicity set from an array of keys.
`multiset_cardinality`	Cardinality of N key-multiplicity sets as the sum of multiplicities.
`multiset_increment`	Increments a key-count set for each key in N multisets/arrays.
`multiset_union`	Union of N key-multiplicity sets as a key-multiplicitysum set.
`object`	Constructs a key-value set from [key,value,key,value,...] array.
`object_cardinality`	Cardinality of a key-value set as the number of keys.
`object_keys`	Keys of a key-value set as an array of keys.
`object_values`	Values of a key-value set as an array of values.
`object_union`	Union of N key-value sets as a key-[value,value,...] set.

Table B.5: JSONMATCH ARRAY FUNCTIONS

Function	Description
`push`	Appends a value to the end of an array.
`append`	Appends an array of values to the end of an array.
`size`	Number of elements in an array.
`sort`	Sort values in an array, possibly on a specified subpart.
`limit`	Truncate array to keep first N values.

Table B.6: JSONMATCH STRING FUNCTIONS

Function	Description
`extract`	Substring matching a regular expression.
`remove_html`	Strips HTML mark-up from a string.
`downcase`	Converts a string to lower case.
`upcase`	Converts a string to upper case.
`length`	Length of string.
`concat`	Concatenate array of strings (with optional separator).
`strip`	Strip specified characters from string.
`latinise`	Replace diacritics with latin character/s.
`normalise_whitespace`	Replace whitespace sequences by a single space.
`split`	Splits a string into a list of words.

Table B.7: JSONMATCH TEXT FUNCTIONS

Function	Description
`entities`	Recognises names of people, places, etc.
`stem`	Replaces words with their Porter stem.
`ngrams`	Returns *n*-grams from a list of words.
`pgrams`	Returns *p*-spectrum of a list of words.
`prefixes`	Counts left substrings of a list of words.
`exclude_words`	Removes supplied 'stopwords' from list.
`include_words`	Restricts words to supplied vocabulary.
`replace_words`	Substitutes words using supplied mapping.
`min_length`	Deletes words below specified length.
`stopwords`	Set of common English words.

Table B.8: JSONMATCH COMPARISON FUNCTIONS

Function	Description
`if`	If argument1 then argument2 else argument3.
`eq`	True if arguments are equal.
`gt`	True if argument1 > argument2.
`lt`	True if argument1 < argument2.
`not`	True if argument is false.
`tfidf`	Calculates TF-IDF for a multiset of words.
`cosine`	Cosine similarity of term-weight vectors.
`kernel`	Kernel (as a distance) on JSON values.

Appendix C

JSONMatch Submission Sifting Workflow

This appendix presents a complete set of REST API method calls for all the higher-order transformations featured in the JSONMatch implementation of the submission sifting workflow. The schematic for this workflow was introduced in Figure 4.15 (p.134) from Chapter 4, where a partial sequence of method calls was given in an abbreviated form for clarity of example. The method calls here include unabbreviated REST API method parameters, although output examples shown are still abbreviated to maintain readability. As a convenience we reproduce the same workflow here as Figure C.1 but in a more compact format that labels transformations as their higher-order parameters (previously depicted as "HO Params." in the Workflow Inputs).

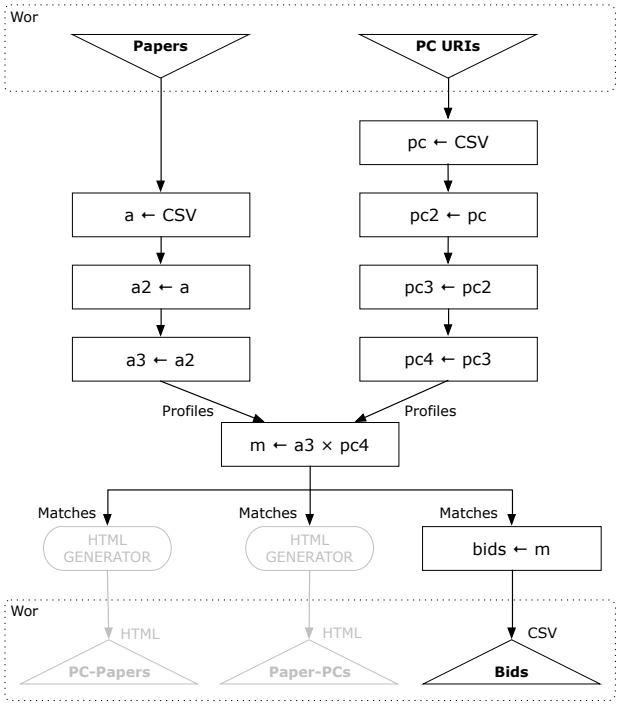


Figure C.1: Higher-order transformations in the *submission sifting* workflow.

Below we describe the transformations involved in the submission sifting workflow under the following four headings corresponding respectively to the top-left branch, top-right branch, joining and bottom-right branches depicted in Figure C.1.

C.1 Profiling Submitted Papers

C.2 Profiling Programme Committee Members

C.3 Profile Matching

C.4 Bid Initialisation

We do not present the HTML Generator steps in the workflow as these are implemented outside of JSONMatch using established web templating techniques.

C.1 Profiling Submitted Papers

The following three transformations compute profiles for each of the submitted papers. The first ingests the titles and abstracts to produce an item per paper, the second "cleans" the text before producing term counts and the third normalises the term counts to produce tf-idf scores for each term in each paper.

Transformation 1 (a ← CSV)**.** The input is a CSV file with rows: `<id>`, `<title>`, `<text>`, i.e. the paper id, paper title and paper abstract text for each paper. This can be loaded into JSONMatch to create a basic relation called `a` (denoting abstracts), with `id` as the item id as specified by the JSONPath expression `"$[0]"`, referring to the 0^{th} element of each input row, using the following HTTP request.

```
POST /relations/a/items
from=csv
id_path=$[0]
is_schema=1
use_schema=1
value=<text of papers csv file>
```

By default JSONMatch will import CSV data, creating each item as an array corresponding to one row in the CSV data. For illustrative purposes, here we use the optional `is_schema` parameter (1=true, 0=false, default of 0) to specify that the CSV file has a heading row consisting of column titles and so that row should not itself produce and item in the output relation. We also the closely related `use_schema=1` parameter to specify that JSONMatch should create an item consisting of a JSON object for each row, using the column titles as the key in each key-value pair in the object. This creates items with the following structure, which can be retrieved from JSONMatch using a `GET /relations/a/items` request to display the first 10 items (optional numeric parameters `count` and `page` allow access to further items).

```
{
  "item" : {
```

```
    "<paper1_id>" : {
      "id" : "<paper1_id>",
      "title" : "<paper1_title>",
      "text" : "<paper1_abstract>"
    },
    "<paper2_id>" : {
      "id" : "<paper2_id>",
      "title" : "<paper2_title>",
      "text" : "<paper2_abstract>"
    },
    ...
    "<paper10_id>" : {
      "id" : "<paper10_id>",
      "title" : "<paper10_title>",
      "text" : "<paper10_abstract>"
    }
  }
}
```

Transformation 2 (a2 ← a)**.** A map transformation now processes the concatenated `title` and `text` string, identified by JSONPath `"$.items[0].title"` and `"$.items[0].text"` respectively, in each input item. This is exactly the same text processing as used in SubSift, but there it was controlled by API parameters; here it is controlled by the nesting of embedded functions in the `item` template below. The processing is essentially a sequence: beginning by converting the text to lowercase, removing HTML tags, splitting the text into an array of words (of 3 to 20 characters in length, using whitespace as the word boundary), removing common English words (i.e. excluding stopwords), replacing this array words with an array of terms of 1-3 words in length (i.e. 1-grams, 2-grams and 3-grams), then (for illustrative purposes, but rather pointlessly) restricts the terms to be any term at all[1], before finally counting the frequency of each term and storing the term-frequency pairs as a multiset (bag) in the `term_n` value of the respective output item.

As paper items (documents) are processed, totals for all the papers (corpus) are accumulated in the relation properties by summing the `term_n` and `document_n` term counts of each item. Exactly how this is acheived is explained shortly.

```
POST /relations/a2/from/a
generator=map
templates={
  "relation": {
    "corpus_n": {},   // bag of term-n count pairs
    "corpus_dt": {},  // bag of term-dt count pairs
    "corpus_N": "$.arguments_size.[0]" // #items in relation a
  },
  "item": {
    "id": "$.items[0].id",
    "title": "$.items[0].title",
    "term_n": ["jm:multiset", // term-n counts for document
      ["jm:include_words",
        ["jm:ngrams",
          ["jm:exclude_words",
```

[1] In Use Case 4 terms are restricted to the ACM Computing Classification System (CCS) lexicon.

```
            ["jm:split",
              ["jm:remove_html",
                ["jm:downcase",
                  ["jm:concat",
                    ["$.items[0].title", " ", "$.items[0].text"]
                  ]
                ]
              ],
              {"minimum_length": 3, "maximum_length": 20,
               "regex": "[\\W]"}
            ],
            {"ignore_case": 0, "exclude": ["jm:stopwords"]}
          ],
          {"nvalues": [1,2,3]} //1-grams, 2-grams, 3-grams
        ],
        {"ignore_case": 0, "include": {}} //i.e. unrestricted!
      ]
    ],
    // document_n is the total no. terms in this document
    "document_n": ["jm:multiset_cardinality", "$.item.term_n"]
  },
  "relation_each_item": {
    // update relation-level counter variables
    "corpus_n": ["jm:multiset_union", "$.relation.corpus_n",
                 "$.item.term_n"],
    "corpus_dt": ["jm:multiset_increment", "$.relation.corpus_dt",
                  "$.item.term_n"]
  }
}
```

For brevity, in the body of the Thesis we abbreviated the `templates` parameter in our examples. In particular, we reduced the number of text pre-processing embedded functions and did not separately show the `relation_each_item` template. The latter is conceptually part of the `relation` parameter but our proof of concept compiler requires this "sub-template" to be explicitly declared even though a more sophisticated compiler implementation could detect dependencies between variables and shield the user from this complexity. In our implementation, the `relation` template is evaluated just once, at the start of the transformation whereas the `relation_each_item` template is evaluated after each `item` is generated – giving the user a way to manually specify which expressions are evaluated at each item and which can just be evaluated once as an initialisation step before any items are generated. In effect, this is a manual optimisation that should ideally be performed by the compiler. Also, as discussed in Chapter 4 transformation involving relations that have global dependencies on items are not guaranteed to be embarrassingly parallel as they require shared read/write access to variables at the relation-level, such as the `corpus_n` and `corpus_dt` counters in the above transformation example.

Below we give real-world example of a single item from `a` and the corresponding output item from `a2` after this transformation for one of the accepted papers from the ECML-PKDD 2012 conference. In reading these examples, observe that JSON-Match serialises JSON object keys in alphabetical order, which can differ from the order specified in the template. Keys in JSON objects are, by definition in the JSON language, unordered but to make it easier for developers to control dependencies on

the evaluation order in higher-order templates, JSONMatch ensures that key-value pairs are processed in the order specified in the template (e.g. `document_n` in the above template depends on `item_n` having already been computed before its own evaluation and so is positioned after `item_n` in the template).

An item from the input relation `a`, as viewed by the HTTP request:

```
GET /relations/a/items/44

{
  "item" : {
    "44" : {
      "id" : "44",
      "text" : "Distinct social networks are interconnected via
bridge users, who play thus a key role when crossing information
is investigated in the context of Social Internetworking
analysis. Unfortunately, not always users make their role of
bridge explicit by specifying the so-called me edge (i.e., the
edge connecting the accounts of the same user in two distinct
social networks), missing thus a potentially very useful
information. As a consequence, discovering missing me edges is an
important problem to face in this context yet not so far
investigated. In this paper, we propose a common-neighbors
approach to detecting missing me edges, which returns good
results in real life settings. Indeed, an experimental campaign
shows both that the state-of-the-art common-neighbors approaches
cannot be effectively applied to our problem and, conversely,
that our approach returns precise and complete results.",
      "title" : "Discovering Links among Social Networks"
    }
  }
}
```

An item from the output relation `a2`, as viewed by the HTTP request:

```
GET /relations/a2/items/44

{
  "item" : {
    "44" : {
      "document_n" : 81,
      "id" : "44",
      "term_n" : {
        "accounts" : 1,
        "analysis" : 1,
        "applied" : 1,
        "approach" : 2,
        "approaches" : 1,
        "art" : 1,
        "bridge" : 2,
        "called" : 1,
        "campaign" : 1,
        "common" : 2,
        "complete" : 1,
```

```
        "connecting" : 1,
        "consequence" : 1,
        "context" : 2,
        "conversely" : 1,
        "crossing" : 1,
        "detecting" : 1,
        "discovering" : 2,
        "distinct" : 2,
        "edge" : 2,
        "edges" : 2,
        "effectively" : 1,
        "experimental" : 1,
        "explicit" : 1,
        "face" : 1,
        "far" : 1,
        "good" : 1,
        "important" : 1,
        "information" : 2,
        "interconnected" : 1,
        "internetworking" : 1,
        "investigated" : 2,
        "key" : 1,
        "life" : 1,
        "links" : 1,
        "make" : 1,
        "missing" : 3,
        "neighbors" : 2,
        "networks" : 3,
        "paper" : 1,
        "play" : 1,
        "potentially" : 1,
        "precise" : 1,
        "problem" : 2,
        "propose" : 1,
        "real" : 1,
        "results" : 2,
        "returns" : 2,
        "role" : 2,
        "settings" : 1,
        "shows" : 1,
        "social" : 4,
        "specifying" : 1,
        "state" : 1,
        "unfortunately" : 1,
        "useful" : 1,
        "user" : 1,
        "users" : 2
      },
      "title" : "Discovering Links among Social Networks"
    }
  }
}
```

For brevity, we cheated slightly in the above example and changed the `nvalues` parameter the `"jm:ngrams"` function call to be `[1]` rather than the `[1,2,3]` shown in the template. Otherwise *2*-grams (e.g. `"distinct social"`, `"social networks"`, `"networks are"`) and, longer still, *3*-grams (e.g. `"distinct social networks"`,

"social networks are", "networks are interconnected") would be added to the already long list of *n*-grams.

The corpus totals are accumulated on the relation properties of a2 as specified by the relation and relation_each_item templates in the templates parameter of the transformation. The relation properties contain a number of system-controlled metadata values, such as "id" and "uri", and so the user-controlled values are stored with the key "value" and, in this example, the associated value is an object (although any type of JSON value can be chosen to suit a particular problem setting). Below we give an abbreviated listing of the relation properties of the output relation a2.

The output relation properties of a2, as viewed by the HTTP request:

```
GET /relations/a2
full=1
{
  "relation": {
    "description" : "...",
    "generator" : "map",
    "id" : "a2",
    "mode" : "public",
    "relation_ids" : ["a"],
    "templates" : {...},
    "type" : "relations",
    "uri" : "...",
    "value" : {
      "corpus_N" : 443,
      "corpus_dt" : {
        "abbreviations" : 1,
        "abilities" : 3,
        "ability" : 19,
        "able" : 36,
        "abnormal" : 3,
        ...
        "zippers" : 1,
        "zones" : 1,
        "zoo" : 1
      },
      "corpus_n" : {
        "abbreviations" : 1,
        "abilities" : 3,
        "ability" : 22,
        "able" : 44,
        "abnormal" : 4,
        ...
        "zippers" : 1,
        "zones" : 1,
        "zoo" : 1
      }
    }
  }
}
```

Transformation 3 (a3 ← a2)**.** This map transformation uses the totals for corpus_n and corpus_dt to normalise each item's term counts and thus calculate the tf-idf

score for each term in each paper as a weighted measure of how discriminating each term is within the paper and across the set of all the submitted papers.

In the submission sifting workflow, profiles are intended to be matched against other profiles and so the corpus statistics are stored in the relation properties again. However, the `corpus_n` and `corpus_dt` values are recalculated after retaining only the top, in this example, 100 highest tf-idf scoring terms for each paper and after removing terms with tf-idf scores less than a threshold of, in this example, 0.01 ; this recalculation would not be required if low scoring terms were not removed or if the profile were not subsequently to be used in a match operation. The `corpus_N` relation property value and the `document_n` value for each paper is also retained (unchanged) for future use in profile matching.

```
POST /relations/a3/from/a2
generator=map
templates={
  "relation": {
    "corpus_n": {},
    "corpus_dt": {},
    "corpus_N": "$.relations[0].corpus_N"
  },
  "item": {
    "id": "$.items[0].id",
    "title": "$.items[0].title",
    "term":
      ["jm:restrict",
        ["jm:limit",
          ["jm:sort",
            ["jm:tfidf", // returns [[term,tfidf,n,tf,idf],...]
              "$.items[0].term_n",
              "$.items[0].document_n",
              "$.relations[0].corpus_dt",
              "$.relations[0].corpus_N"
            ],
            {"index": 1}  // 2nd elem. in [term,tfidf,n,tf,idf]
          ],
          {"limit": 100}  // only keep first 100 array elements
        ],
        ["jm:gt", "$.item[1]", 0.01]  // discards if tdidf < 0.01
      ],
    "document_n": "$.items[0].document_n"
  },
  "relation_each_item": {
    "corpus_n": ["jm:multiset_union", "$.relation.corpus_n",
      // select every 1st and 3rd elements of [term,tfidf,n,tf,idf]
      ["jm:project", "$[*][0,2]", "$.item.term"]
    ],
    "corpus_dt": ["jm:multiset_increment", "$.relation.corpus_dt",
      // select every 1st element of [term,tfidf,n,tf,idf]
      ["jm:project", "$[*][0]", "$.item.term"]
    ]
  }
}
```

An item from the output relation `a3`, as viewed by the HTTP request:

```
GET /relations/a2/items/44

{
  "item" : {
    "44" : {
      "document_n" : 81,
      "id" : "44",
      "term" : [
        // "<term>", <tfidf>, <n>, <tf>, <idf>
        ["missing", 0.2144, 3, 0.0370, 5.7911],
        ["returns", 0.1923, 2, 0.0246, 7.7911],
        ["bridge", 0.1923, 2, 0.0246, 7.7911],
        ["investigated", 0.1597, 2, 0.0246, 6.4692],
        ["distinct", 0.1532, 2, 0.0246, 6.2062],
        ["social", 0.1512, 4, 0.0493, 3.0632],
        ["edge", 0.1429, 2, 0.0246, 5.7911],
        ["neighbors", 0.1285, 2, 0.0246, 5.2062],
        ...
        ["analysis", 0.0298, 1, 0.0123, 2.4161],
        ["approaches", 0.0294, 1, 0.0123, 2.3817],
        ["experimental", 0.0289, 1, 0.0123, 2.3482],
        ["real", 0.0200, 1, 0.0123, 1.6212],
        ["propose", 0.0161, 1, 0.0123, 1.3073]
      ],
      "title" : "Discovering Links among Social Networks"
    }
  }
}
```

As well as obviously omitting many of the terms from the `term` array (and inserting a comment to label the column headings) we have also post-edited the numbers for brevity, truncating them after three decimal places. Even so, it can be seen that commonly used terms like "analysis", "approaches" and "experimental" appear near the bottom of the descending tf-idf sorted array of terms, whereas terms like "missing", "returns" and "bridge" are ranked as the most discriminating terms for this paper. The fact that common (in this domain) terms have been retained suggests that the threshold of 0.01 should be raised or that fewer than 100 terms should be kept.

C.2 Profiling Programme Committee (PC) Members

This part of the submission sifting workflow has already been described in abbreviated form in Section 4.4.3 (p.125) of Chapter 4. Here we present the full higher-order `templates` parameters for each transformation but, for brevity, we do not include the keywords processing outlined in our earlier presentation.

Three of the four transformations in the dataflow that computes the profiles of PC members are almost identical to the three transformations we used to profile the submitted papers above, differing only in the names of the relations and the origin of the text to be compared. The input to the dataflow described in this section is not the text to be profiled, as previously, but the URLs of the PC members' DBLP author pages. Consequently a fourth transformation precedes the three already described, in order to fetch the HTML text to be profiled from the DBLP author page specified by

each URL.

Transformation 4 (pc ← CSV)**.** A CSV file with rows, `<pc_member_name>`, `<url>`, is loaded into JSONMatch to create a basic relation called `pc`, with `pc_member_name` selected as the item id by `"$[0]"`, in the following request.

```
POST /relations/pc/items
from=csv
id_path=$[0]
value=<text of PC csv file>
```

The items in the resultant relation have the following format, where the ids are generated automatically from the string at JSONPath `"$[0]"`, converting the string to be web-friendly and a legal JSON object key.

```
{
  "item": {
    <pc_member1_id>: [
      <pc_member1_name>,
      <pc_member1_url>
    ],
    <pc_member2_id>: [
      <pc_member2_name>,
      <pc_member2_url>
    ],
    ...
    <pc_memberM_id>: [
      <pc_memberM_name>,
      <pc_memberM_url>
    ]
  }
}
```

An item from the output relation `pc` for the PC members of the ECML-PKDD 2012 conference, as viewed by the HTTP request:

```
GET /relations/pc/items/Ad_Feelders

{
  "item": {
    "Ad_Feelders": [
      "Ad Feelders",
      "http://dblp.uni-trier.de/pers/hd/f/Feelders:Ad.html"
   ]
  }
}
```

For the ECML-PKDD 2012 conference we did not use the DBLP author pages directly, instead we opted to fetch them indirectly using a standalone web service developed as part of SubSift. This web service extracts just the publication titles

from the full HTML text of a DBLP author page and returns a plain text string with one publication title per line. This is an optimisation to remove commonly occurring terms like "Copyright" and "DBLP", thereby reducing the quantity of text to be processed in calculating profiles and matches. However, if the raw HTML were used then the tf-idf weighting would assign low scores to such terms and they would not contribute significantly (if at all in some cases) to the overall match scores. So, the urls actually used were in the following format.

```
http://subsift.ilrt.bris.ac.uk/demo/dblp_extract?uri=<pc_member_url>
```

In constructing these urls the `<pc_member_url>` must be URL escaped in the standard way (e.g. spaces are represented as `%20`, colon as `%3A`, and so on) so that, for example, the above url for Ad Feelders is encoded as follows.

```
http%3A%2F%2Fdblp.uni-trier.de%2Fsearch%2Fauthor%3Fauthor%3DAd%2520Feelders
```

Transformation 5 (pc2 ← pc)**.** A map transformation now converts the array representing each PC member in relation `pc` to an object with three key-value pairs in relation `pc2` using the HTTP request shown below.

```
POST /relations/pc2/from/pc
generator=map
templates={
  "item": {
    "name": "$.items[0][0]",
    "url": "$.items[0][1]",
    "text": ["jm:http_get", "$.items[0][1]"]
  }
}
```

The third key-value pair, with key `text`, is assigned the HTML text fetched from each PC member's DBLP author page by embedded function `jm:http_get`, which takes a url as its single argument.

An item from the output relation `pc2`, as viewed by the HTTP request:

```
GET /relations/pc3/from/pc2

{
  "item": {
    "Ad_Feelders": {
      "name": "Ad Feelders",
      "text": "<html><head><title>A. J. Feelders</title>...</html>",
      "url": "http://dblp.uni-trier.de/pers/hd/f/Feelders:Ad.html"
    }
  }
}
```

Transformation 6 (pc3 ← pc2)**.** This transformation is almost identical Transformation 2 in profiling the submitted papers. Apart from having different input and output relation names, the minor differences are that here we have `name` instead of `id`, `url` instead of `title`, and there is no need to concatenate a title with the `text` string before processing it.

```
POST /relations/pc3/from/pc2
generator=map
templates={
  "relation": {
    "corpus_n": {},   // bag of term-n count pairs
    "corpus_dt": {},  // bag of term-dt count pairs
    "corpus_N": "$.arguments_size.[0]" // #items in relation a
  },
  "item": {
    "name": "$.items[0].name",
    "url": "$.items[0].url",
    "term_n": ["jm:multiset", // term-n counts for document
      ["jm:include_words",
        ["jm:ngrams",
          ["jm:exclude_words",
            ["jm:split",
              ["jm:remove_html",
                ["jm:downcase", "$.items[0].text"]
              ],
              {"minimum_length": 3, "maximum_length": 20,
               "regex": "[\\W]"}
            ],
            {"ignore_case": 0, "exclude": ["jm:stopwords"]}
          ],
          {"nvalues": [1,2,3]} //1-grams, 2-grams, 3-grams
        ],
        {"ignore_case": 0, "include": {}} //i.e. unrestricted!
      ]
    ],
    // document_n is the total no. terms in this document
    "document_n": ["jm:multiset_cardinality", "$.item.term_n"]
  },
  "relation_each_item": {
    // update relation-level counter variables
    "corpus_n": ["jm:multiset_union", "$.relation.corpus_n",
                 "$.item.term_n"],
    "corpus_dt": ["jm:multiset_increment", "$.relation.corpus_dt",
                  "$.item.term_n"]
  }
}
```

The form of the output relation is identical to that from Transformation 2 except for the key name differences already mentioned.

Transformation 7 (pc4 ← pc3)**.** This transformation is almost identical to Transformation 3 in profiling the submitted papers. Again, two of the key names differ but the remainder of the `templates` parameter is identical. Similarly, the form of the output relation will be almost the same as for Transformation 3.

```
POST /relations/pc4/from/pc3
generator=product
templates={
  "relation": {
    "corpus_n": {},
    "corpus_dt": {},
    "corpus_N": "$.relations[0].corpus_N"
  },
  "item": {
    "name": "$.items[0].name",
    "url": "$.items[0].url",
    "term":
      ["jm:restrict",
        ["jm:limit",
          ["jm:sort",
            ["jm:tfidf", // returns [[term,tfidf,n,tf,idf],...]
              "$.items[0].term_n",
              "$.items[0].document_n",
              "$.relations[0].corpus_dt",
              "$.relations[0].corpus_N"
            ],
            {"index": 1}  // 2nd elem. in [term,tfidf,n,tf,idf]
          ],
          {"limit": 100}  // only keep first 100 array elements
        ],
        ["jm:gt", "$.item[1]", 0.01]  // discards if tdidf < 0.01
      ],
    "document_n": "$.items[0].document_n"
  },
  "relation_each_item": {
    "corpus_n": ["jm:multiset_union", "$.relation.corpus_n",
      // select every 1st and 3rd elements of [term,tfidf,n,tf,idf]
      ["jm:project", "$[*][0,2]", "$.item.term"]
    ],
    "corpus_dt": ["jm:multiset_increment", "$.relation.corpus_dt",
      // select every 1st element of [term,tfidf,n,tf,idf]
      ["jm:project", "$[*][0]", "$.item.term"]
    ]
  }
}
```

C.3 Profile Matching

Transformation 8 (m ← a3 × pc4)**.** The left and right branches of the dataflow now come together in a match transformation using the product generator to process pairs composed of an item from `a3` and an item from `pc4`. The pairs are accessed in the templates as `$.items`, referring to `$.items[0]` and `$.items[1]` as the item from `a3` and `pc4` respectively.

```
POST /relations/m/from/a3/pc4
generator=product
templates={
  "relation": {
    "corpus_n": ["jm:multiset_union",
```

```
      "$.relations[0].corpus_n",
      "$.relations[1].corpus_n"
    ],
    "corpus_dt": ["jm:multiset_union",
      "$.relations[0].corpus_dt",
      "$.relations[1].corpus_dt"
    ]
  },
  "item": {
    "title": "$.items[0].title",
    "id": "$.items[0].id",
    "name": "$.items[1].name",
    "url": "$.items[1].url",
    "items": "$.items[*].term",
    "matches":
    ["jm:cosine",
      ["jm:tfidf",
        ["jm:object", ["jm:project", "$[*][0,2]", "$.items[0].term"]],
        "$.items[0].document_n",
        "$.relation.corpus_dt",
        "$.relation.corpus_n"
      ],
      ["jm:tfidf",
        ["jm:object", ["jm:project", "$[*][0,2]", "$.items[1].term"]],
        "$.items[1].document_n",
        "$.relation.corpus_dt",
        "$.relation.corpus_n"
      ]
    ]
  }
}
```

As a product transformation the number of items in the output relation can be large. For example, the 311 accepted ECML-PKDD papers and 125 PC members produces 38,875 output items. The number of items in the output relation `m` can be accessed, including during the transformation to monitor progress of large calculations, using the following query.

```
GET /relations/m/count

{
   "count" : 38875
}
```

Ranges of items in a relation can be accessed by adding a REST parameter `count` to specific the number of items and `page` to specify which "page" to start at, where the page size is `count` (the default is 10). When both are set to 1 then items can be accessed individually. Alternatively, individual items can be accessed by their identifier. To view the list of all identifiers in a relation, the `index` method can be used as follows.

```
GET /relations/m/index
```

```
{
   "id" : [
      "100_x_Ad_Feelders",
      "100_x_Aditya_Menon",
      "100_x_Albert_Bifet",
      ...
      "100_x_Zhi-Hua_Zhou",
      "101_x_Ad_Feelders",
      "101_x_Aditya_Menon",
      "101_x_Albert_Bifet",
      ...
      "101_x_Zhi-Hua_Zhou",
      "102_x_Ad_Feelders",
      "102_x_Aditya_Menon",
      "102_x_Albert_Bifet",
      ...
}
```

To suit our submission sifting use case we have included a lot of metadata in each item of m, for example every item includes the full ranked list of term-tfidf pairs (i.e. the profile) for the PC member and the paper being compared and so each PC member's profile will be repeated 125 times and each paper's 311 times in our earlier example. This is inefficient in terms of storage but can be efficient if the data in the "individual" item contains all the data for a particular query in a web application. In database terminology, this is a materialised view of the data customised for a particular application. However, if an application does not require such data at speed then it could be omitted from `m` and accessed by a separate query to profile relations instead.

An example of a item, `102_x_Ad_Feelders`, in the output relation `m` is shown below. The id `102_x_Ad_Feelders` was constructed automatically from the ids of the pair of input items, by joining them with `"_x_"`.

```
GET /relations/m/items/102_x_Ad_Feelders

{
  "item" : {
    "102_x_Ad_Feelders" : {
      "id" : "102",
      "items" : [
        [
          // a3: "<term>", <tfidf>, <n>, <tf>, <idf>
          ["quantitative", 0.1688, 4, 0.0242, 6.9657],
          ["rule", 0.1610, 7, 0.0424, 3.7958],
          ["attributes", 0.1504, 5, 0.0303, 4.9657],
          ...
          ["performance", 0.0104, 1, 0.0060, 1.7178]
        ],
        [
          // pc4: "<term>", <tfidf>, <n>, <tf>, <idf>
          ["monotonicity", 0.1281, 4, 0.0191, 6.6958],
          ["monotone", 0.1070, 5, 0.0239, 4.4734],
          ["influences", 0.1045, 3, 0.0143, 7.2807],
          ...
          ["discovering", 0.0100, 1, 0.0047, 2.0909]
```

```
        ]
      ],
      "matches" : [
        0.2966,  // cosine similarity score
        [
          // "<term>", <contribution>
          ["classification", 48],
          ["mining", 40],
          ["bayesian", 20],
          ["classifiers", 10],
          ["model", 6],
          ["parameter", 4],
          ["categorical", 3],
          ["models", 3],
          ["prior", 2],
          ["association", 2],
          ["subgroup", 1],
          ["naive", 1]
        ]
      ],
      "name" : "Ad Feelders",
      "title" : "A Bayesian approach for classification rule ...",
      "url" : "http://dblp.uni-trier.de/pers/hd/f/Feelders:Ad.html"
    }
  }
}
```

The first element of the array `items` in the above is the profile data for the paper and the second item is the profile data () for the PC member. These profile data are based on statistics calculated over the combined vocabulary (set of terms) from both papers and PC members (which will be different to the statistics calculated separately for the input profiles where only one vocabulary is used). The most important part of the output is the `matches` pair of cosine similarity score (0.2966) and the following ranked list of contributing terms, each with a figure indicating their contribution to the overall score (the highest being `"classification"` with a contribution of 48 out of 140 (i.e. out of $48+40+20+10+6+4+3+3+2+2+1+1$).

C.4 Bid Initialisation

Transformation 9 (bids ← m)**.** The final transformation computes a relation in which every item consists of just the PC member name, the paper id, and an initial bid of 0-3 which is calculated by thresholding the cosine similarity score according to manually chosen thresholds.

```
POST /relations/bids/from/m
generator=map
templates={
  "item": [
    "$.items[0].name",
    "$.items[0].id",
    // bid 3 (eager), 2 (willing), 1 (at a pinch) and 0 (no bid)
```

```
      ["jm:if", ["jm:gt", "$.items[0].matches[0]", 0.2], 3,
        ["jm:if", ["jm:gt", "$.items[0].matches[0]", 0.1], 2,
          ["jm:if", ["jm:gt", "$.items[0].matches[0]", 0.05], 1,
            0
          ]
        ]
      ]
    ]
  }
```

The output relation is intended to be downloaded as CSV by an HTTP GET query `/relations/bids/items.csv` for subsequent upload into a conference management system.

Bibliography

[Abb11] Dawood Abbas. Optimal allocation of student projects and markers. Masters thesis, Department of Computer Science, University of Bristol, September 2011.

[ABJ + 04] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: an extensible system for design and execution of scientific workflows. In *Scientific and Statistical Database Management, 2004. Proceedings. 16th International Conference on.*, pages 423–424, 2004.

[AF10] Tarek Abudawood and Peter A. Flach. Exploiting the high predictive power of multi-class subgroups. *Journal of Machine Learning Research - Proceedings Track*, 13:177–192, 2010.

[ATT + 10] S. Ananiadou, P. Thompson, J. Thomas, T. Mu, S. Oliver, M. Rickinson, Y. Sasaki, D. Weissenbacher, and J. McNaught. Supporting the education evidence portal via text mining. *Philosophical Transcations of the Royal Society A*, 368(1925):3829–3844, 2010.

[BAdR06] Krisztian Balog, Leif Azzopardi, and Maarten de Rijke. Formal models for expert finding in enterprise corpora. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 43–50, New York, NY, USA, 2006. ACM.

[BBB + 13] L Bornmann, BF Bowman, J Bauer, W Marx, H Schier, and M Palzenberger. Standards for using bibliometrics in the evaluation of research institutes. *Next Generation Metrics*, 2013.

[BCC03] Rohan Baxter, Peter Christen, and Tim Churches. A comparison of fast blocking methods for record linkage. In *Proceedings of the 2003 ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, pages 25–27, Washington, DC, 2003.

[BCG + 12] Khalid Belhajjame, Oscar Corcho, Daniel Garijo, Jun Zhao, Paolo Missier, David R. Newman, Raul Palma, Sean Bechhofer, Esteban Garcia Cuesta, Jose Manuel Gomez-Perez, Graham Klyne, Kevin Page, Marco Roos, José Enrique Ruiz, Stian Soiland-Reyes, Lourdes

Verdes-Montenegro, David De Roure, and Carole Goble. Workflow-centric research objects: A first class citizen in the scholarly discourse. In Alexander García Castro, Christoph Lange, Frank van Harmelen, and Benjamin Good, editors, *2nd Workshop on Semantic Publishing (SePublica)*, number 903 in CEUR Workshop Proceedings, pages 1–12, Aachen, 2012.

[BDK+04] Aziz A. Boxwala, Meghan Dierks, Maura Keenan, Susan Jackson, Robert Hanscom, David W. Bates, and Luke Sato. Review paper: Organization and representation of patient safety data: Current status and issues around generalizability and scalability. *Journal of the American Medical Informatics Association: JAMIA*, 11(6):468–478, 2004.

[Bec03] D. Beckett. RDF/XML syntax specification (revised), 1 2003.

[BFH+10] Remco R. Bouckaert, Eibe Frank, Mark A. Hall, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. WEKA–experiences with a Java open-source project. *Journal of Machine Learning Research*, 11:2533–2541, 2010.

[BG05] Indrajit Bhattacharya and Lise Getoor. Relational clustering for multi-type entity resolution. In *MRDM '05: Proceedings of the 4th international workshop on Multi-relational mining*, pages 3–12, New York, NY, USA, 2005. ACM Press.

[BG06] Indrajit Bhattacharya and Lise Getoor. A latent Dirichlet model for unsupervised entity resolution. In *6th SIAM Conference on Data Mining (SDM-2006)*, Bethesda, MD, 2006.

[BGM04] Dan Brickley, R.V. Guha, and Brian McBride. RDF vocabulary description language 1.0: RDF Schema, 2 2004.

[BHCNm99] Chumki Basu, Haym Hirsh, William W. Cohen, and Craig Nevill-manning. Recommending papers by mining the web. In *In: Proceedings of the IJCAI99 Workshop on Learning about Users*, pages 1–11, 1999.

[BHJ+10] Dominik Benz, Andreas Hotho, Robert Jäschke, Beate Krause, Folke Mitzlaff, Christoph Schmitz, and Gerd Stumme. The social bookmark and publication management system bibsonomy. *The VLDB Journal*, 19(6):849–875, December 2010.

[BHW98] Uta Bohnebeck, Tamás Horváth, and Stefan Wrobel. Term comparisons in first-order similarity measures. In David Page, editor, *ILP*, volume 1446 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 1998.

[BJZ02] J. Brixey, T.R. Johnson, and J. Zhang. Evaluating a medical error taxonomy. *Proceedings of the American Medical Informatics Association (AMIA) Symposium*, 2002.

[BL98] T. Berners-Lee. IETF RFC 2396, Uniform Resource Identifiers (URI): Generic syntax, 8 1998.

[BL01] Salem Benferhat and Jérôme Lang. Conference paper assignment. *International Journal of Intelligent Systems*, 16(10):1183–1192, 2001.

[BLHL01] T. Berners-Lee, J. Hendler, and O.T. Lassila. The Semantic Web. *Scientific American*, May 2001.

[BM04] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition*. W3C, W3C Recommendation 28 October 2004 edition, October 2004.

[BMPQ04] Philip A. Bernstein, Sergey Melnik, Michalis Petropoulos, and Christoph Quix. Industrial-strength schema matching. *SIGMOD Rec.*, 33(4):38–43, December 2004.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsson Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3 – 48, 1995.

[BOHG13] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109 – 132, 2013.

[BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.*, 30(1-7):107–117, April 1998.

[BPSM + 06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, August 2006.

[BRGC + 12] Khalid Belhajjame, Marco Roos, Esteban Garcia-Cuesta, Graham Klyne, Jun Zhao, David De Roure, Carole Goble, Jose Manuel Gomez-Perez, Kristina Hettne, and Aleix Garrido. Why workflows break - understanding and combating decay in taverna workflows. In *Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science)*, E-SCIENCE '12, pages 1–9, Washington, DC, USA, 2012. IEEE Computer Society.

[BTN + 10] Jiten Bhagat, Franck Tanoh, Eric Nzuobontane, Thomas Laurent, Jerzy Orlowski, Marco Roos, Katy Wolstencroft, Sergejs Aleksejevs, Robert Stevens, Steve Pettifer, Rodrigo Lopez, and Carole A. Goble. BioCatalogue: a universal catalogue of web services for the life sciences. *Nucleic Acids Research*, 38:689–694, 2010.

[CBO + 12] Michael A. Covington, Roberto Bagnara, Richard A. O'Keefe, Jan Wielemaker, and Simon Price. Coding guidelines for Prolog. *TPLP*, 12(6):889–927, 2012.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 2011.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(2):56–68, June 1940.

[CM05] Aron Culotta and Andrew McCallum. Joint deduplication of multiple record types in relational data. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 257–258, New York, NY, USA, 2005. ACM.

[Cod69] E. F. Codd. Derivability, redundancy, and consistency of relations stored in large data banks. Technical Report JR599, IBM, San Jose, 1969.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems, Vol. 4, No. 4, December 1979.*, 4(4):397–434, December 1979.

[Cod90] E. F. Codd. *The Relational Model for Database Management, Version 2*. Addison Wesley, 1990.

[Cou13] Raphael Couturier, editor. *Designing Scientific Applications on GPUs*. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series. Chapman and Hall/CRC, November 2013.

[CvHH+01] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. DAML+OIL reference description, 12 2001.

[CWB07] Kevin R Coombes, Jing Wang, and Keith A Baggerly. Microarrays: retracing steps. *Nat Med*, 13(11):1276–1277, 11 2007.

[CZB11] Laurent Charlin, Richard S. Zemel, and Craig Boutilier. A framework for optimizing paper matching. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2011.

[Dat91] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

[DBL12] *8th IEEE International Conference on E-Science, e-Science 2012, Chicago, IL, USA, October 8-12, 2012*. IEEE Computer Society, 2012.

[dFE07] Claudia d'Amato, Nicola Fanizzi, and Floriana Esposito. Induction of optimal semantic semi-distances for clausal knowledge bases. In Hendrik Blockeel, Jan Ramon, Jude W. Shavlik, and Prasad Tadepalli, editors, *ILP*, volume 4894 of *Lecture Notes in Computer Science*, pages 29–38. Springer, 2007.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[DKL08] Hongbo Deng, Irwin King, and Michael R. Lyu. Formal models for expert finding on dblp bibliography data. In *In ICDM*, pages 163–172, 2008.

[DMD+03] AnHai Doan, Jayant Madhavan, Robin Dhamankar, Pedro Domingos, and Alon Halevy. Learning to match ontologies on the Semantic Web. *The VLDB Journal*, 12(4):303–319, November 2003.

[DNH04] AnHai Doan, Natalya F. Noy, and Alon Y. Halevy. Introduction to the special issue on semantic integration. *SIGMOD Rec.*, 33(4):11–13, 2004.

[DR13] David De Roure. Towards computational research objects. In *Proceedings of the 1st International Workshop on Digital Preservation of Research Methods and Artefacts*, DPRMA '13, pages 16–19, New York, NY, USA, 2013. ACM.

[DRGS09] David De Roure, Carole Goble, and Robert Stevens. The design and realisation of the virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25(5):561–567, 2009.

[DYX10] YueHua Ding, Kui Yi, and RiHua Xiang. Design of paper duplicate detection system based on lucene. *Wearable Computing Systems, Asia-Pacific Conference on*, pages 36–39, 2010.

[EBB+04] Jerome Euzenat, Thanh Le Bach, Jesus Barrasa, Paolo Bouquet, Jan De Bo, Rose Dieng, Marc Ehrig, Manfred Hauswirth, Mustafa Jarrar, Ruben Lara, Diana Maynard, Amedeo Napoli, Giorgos Stamou, Heiner Stuckenschmidt, Pavel Shvaiko, Sergio Tessaris, Sven Van Acker, and Ilya Zaihrayeu. State of the art on ontology alignment. Kweb eu-ist-2004-507482 project report, INRIA, 2004.

[ECM13] ECMA International. *ECMA-404 The JSON Data Interchange Standard*, 1st edition edition, October 2013.

[EHT10] Erik Elmroth, Francisco Hernández, and Johan Tordsson. Three fundamental dimensions of scientific workflow interoperability: Model of computation, language, and execution environment. *Future Gener. Comput. Syst.*, 26:245–256, February 2010.

[EN06] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 5th edition, March 2006.

[EPM13] Thomas Erl, Ricardo Puttini, and Zaigham Mahmood. *Cloud Computing: Concepts, Technology & Architecture*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2013.

[FAM00] Katerina T. Frantzi, Sophia Ananiadou, and Hideki Mima. Automatic recognition of multi-word terms: the c-value/nc-value method. *Int. J. on Digital Libraries*, 3(2):115–130, 2000.

[FHM05] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspaces: A new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, December 2005.

[FHM08] Michael J. Franklin, Alon Y. Halevy, and David Maier. A first tutorial on dataspaces. *PVLDB*, 1(2):1516–1517, 2008.

[Fla12] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012.

[FLM98] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the World-Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, 1998.

[Fos95] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Parallel programming / scientific computing. Addison-Wesley, 1995.

[FS69] Ivan P. Fellegi and Alan B. Sunter. A theory for record linkage. *American Statistical Association Journal*, 64(328):1183–1210, December 1969.

[FSG + 09] Peter A. Flach, Sebastian Spiegler, Bruno Golénia, Simon Price, John Guiver Ralf Herbrich, Thore Graepel, and Mohammed J. Zaki. Novel tools to streamline the conference review process: Experiences from SIGKDD'09. *SIGKDD Explorations*, 11(2):63–67, December 2009.

[FT02] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2:115–150, May 2002.

[FZ07] Hui Fang and ChengXiang Zhai. Probabilistic models for expert finding. In *Proceedings of the 29th European Conference on IR Research*, ECIR'07, pages 418–430, Berlin, Heidelberg, 2007. Springer-Verlag.

[GAB + 12] Daniel Garijo, Pinar Alper, Khalid Belhajjame, Óscar Corcho, Yolanda Gil, and Carole A. Goble. Common motifs in scientific workflows: An empirical analysis. In *eScience*, pages 1–8, 2012.

[GBL98] C. Lee Giles, Kurt Bollacker, and Steve Lawrence. CiteSeer: An automatic citation indexing system. In *Proceedings of the 3rd ACM Conference on Digital Libraries*, pages 89–98, Pittsburgh, PA, 1998.

[GF05] Elias Gyftodimos and Peter A. Flach. Combining bayesian networks with higher-order data representations. In *Proceedings of the 6th International Symposium on Intelligent Data Analysis (IDA'06)*, pages 145–157. Springer-Verlag, September 2005.

[GGR+03] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: Learning Document Type Descriptors from XML document collections. *Data Mining and Knowledge Discovery*, 7(1):23–56, January 2003.

[GLF04] Thomas Gaertner, John W. Lloyd, and Peter A. Flach. Kernels and distances for structured data. *Machine Learning*, 57(3):205–232, December 2004.

[GMN84] Herve Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, 1984.

[GNC+09] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh Srinivasan, and Utkarsh Srivastava. Building a high-level dataflow system on top of map-reduce: The pig experience. In *in VLDB '09: Proceedings of the VLDB Endowment*, pages 1414–1425, 2009.

[GNT10] Jeremy Goecks, Anton Nekrutenko, and James Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

[Gös07] Stefan Gössner. JSONPath - XPath for JSON, February 2007.

[GVL10] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, 2010.

[Ham12] Ruth Hambley. ECMLiPlanner: The intelligent scheduling application. Masters thesis, Department of Computer Science, University of Bristol, September 2012.

[Har93] Donna K. Harman. The first text retrieval conference (trec-1) rockville, md, u.s.a., 4–6 november, 1992. *Inf. Process. Manage.*, 29(4):411–414, July 1993.

[Har09] Stevan Harnad. Open access scientometrics and the uk research assessment exercise. *Scientometrics*, 79(1):147–156, April 2009.

[HFM06] Alon Halevy, Michael Franklin, and David Maier. Principles of dataspace systems. In *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '06, pages 1–9, New York, NY, USA, 2006. ACM.

[HG04] Jonathan Hayes and Claudio Gutierrez. Bipartite graphs as intermediate model for RDF. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *The Semantic Web – ISWC 2004, Third*

International Semantic Web Conference, Hiroshima, Japan, Proceedings., volume 3298 of *Lecture Notes in Computer Science*, pages 47–61, Berlin / Heidelberg, November 2004. Springer-Verlag.

[HKS+07] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In *Proceedings of the 4th International Conference on Data Integration in the Life Sciences*, DILS'07, pages 105–121, Berlin, Heidelberg, 2007. Springer-Verlag.

[HP06] Seth Hettich and Michael J. Pazzani. Mining for proposal reviewers: lessons learned at the national science foundation. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 862–871, New York, NY, USA, 2006. ACM.

[HWS+06] Duncan Hull, Katherine Wolstencroft, Robert Stevens, Carole Goble, Matthew Pocock, Peter Li, and Thomas Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34:729–732, July 2006.

[HZC10] Yanan Hao, Yanchun Zhang, and Jinli Cao. Web services discovery and rank: An information retrieval approach. *Future Generation Computer Systems*, 26(8):1053 – 1062, 2010.

[Int05] International Federation of Library Associations and Institutions. IFLANET - digital libraries: Metadata resources, 7 2005.

[Jar89] Mathew A. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida. *Journal of the American Statistical Association*, 84(406):414–420, June 1989.

[KC04] Graham Klyne and Jeremy J. Carroll. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. W3C, w3c recommendation 10 february 2004 edition, 10 2004.

[KDN+10] Yoshinobu Kano, Paul Dobson, Mio Nakanishi, Jun ichi Tsujii, and Sophia Ananiadou. Text mining meets workflow: linking u-compare with taverna. *Bioinformatics*, 26(19):2486–2487, 2010.

[Kel12] Thomas Kelly. Timetable scheduling by profiling through SubSift. Masters thesis, Department of Computer Science, University of Bristol, September 2012.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46(5):604–632, September 1999.

[KM77] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[KM01] M. Koivunen and E. Miller. W3C Semantic Web Activity, 12 2001.

[KW00] Mathias Kirsten and Stefan Wrobel. Extending k-means clustering to first-order representations. In James Cussens and Alan M. Frisch, editors, *ILP*, volume 1866 of *Lecture Notes in Computer Science*, pages 112–129. Springer, 2000.

[LA03] Bertram Ludäscher and Ilkay Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks. Technical report, San Diego Supercomputer Center, 2003.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, August 2006.

[LB10] Danielle H. Lee and Peter Brusilovsky. Social networks and interest similarity: the case of citeulike. In *Proceedings of the 21st ACM conference on Hypertext and hypermedia*, HT '10, pages 151–156, New York, NY, USA, 2010. ACM.

[LBG99] Steve Lawrence, Kurt Bollacker, and C. Lee Giles. Autonomous citation matching. In *Proceedings of the 3rd International Conference on Autonomous Agents*, pages 392–393, New York, NY, May 1999. ACM Press.

[LD97] Nada Lavrac and Saso Dzeroski, editors. *Inductive Logic Programming, 7th International Workshop, ILP-97, Prague, Czech Republic, September 17-20, 1997, Proceedings*, volume 1297 of *Lecture Notes in Computer Science*. Springer, 1997.

[Ley02] Michael Ley. The dblp computer science bibliography: Evolution, research issues, perspectives. In *Proceedings of the 9th International Symposium on String Processing and Information Retrieval*, SPIRE 2002, pages 1–10, London, UK, 2002. Springer-Verlag.

[LGS07] Jon Lathem, Karthik Gomadam, and Amit P. Sheth. SA-REST and (S)mashups: Adding semantics to RESTful services. *International Conference on Semantic Computing*, 11(6):469–476, 2007.

[Llo02] John W. Lloyd. Higher-order computational logic. In Antonis C. Kakas and Fariba Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 105–137. Springer, 2002.

[Llo03] John W. Lloyd. *Logic and Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[LMR05] X. Li, P. Morie, and D. Roth. Semantic integration in text: From ambiguous names to identifiable entities. *AI Magazine: Special Issue on Semantic Integration*, 26(1), 2005.

[LRL+12] Richard Littauer, Karthik Ram, Bertram Ludäscher, William Michener, and Rebecca Koskela. Trends in Use of Scientific Workflows: Insights from a Public Repository and Recommendations for Best Practice. *International Journal of Digital Curation*, 7(2):92–100, October 2012.

[Mae02] Alexander Maedche. *Ontology Learning for the Semantic Web*. Kluwer Academic Publishers, 2002.

[Mai83] David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md., USA, 1983.

[MB05] Timothy M. McPhillips and Shawn Bowers. An approach for pipelining nested collections in scientific workflows. *SIGMOD Rec.*, 34(3):12–17, September 2005.

[MM03] F. Manola and E. Miller. RDF primer, 1 2003.

[MM07] David Mimno and Andrew McCallum. Expertise modeling for matching papers with reviewers. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '07, pages 500–509, New York, NY, USA, 2007. ACM.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Piscataway, NJ, USA, 1989. IEEE Press.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, July 1991.

[MS02] Alexander Maedche and Steffen Staab. Measuring similarity between ontologies. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 251–263, London, UK, 2002. Springer-Verlag.

[MS05] Bradley Malin and Latanya Sweeney. ENRES: A semantic framework for entity resolution modelling. Technical Report CMU-ISRI-05-134, Carnegie Mellon University, Institute for Software Research International, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA USA 15213-3890, 2005.

[MvH12] Deborah L. McGuinness and Frank van Harmelen. OWL 2 Web Ontology Language document overview, December 2012.

[NC97] Shan-Hwei Nienhuys-Cheng. Distance between herbrand interpretations: A measure for approximations to a target concept. In Lavrac and Dzeroski [LD97], pages 213–226.

[New01] M. E. J. Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2):404–409, January 2001.

[NKAJ59] H.B. Newcombe, J.M. Kennedy, S.J. Axford, and A.P. James. Automatic linkage of vital records. *Science*, 130:954–959, 1959.

[Noy04] Natalya F. Noy. Semantic integration: a survey of ontology-based approaches. *SIGMOD Rec.*, 33(4):65–70, December 2004.

[OA06] Naoaki Okazaki and Sophia Ananiadou. Building an abbreviation dictionary using a term recognition approach. *Bioinformatics*, 22:3089–3095, December 2006.

[OGA + 06] Thomas Oinn, Mark Greenwood, Matthew Addis, Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, Peter Li, Phillip Lord, Matthew Pocock, Martin Senger, Robert Stevens, Anil Wipat, and Christopher Wroe. Taverna: lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, August 2006.

[ORS + 08] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1099–1110, New York, NY, USA, 2008. ACM.

[OVvdA + 07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Sci. Comput. Program.*, 67(2-3):162–198, July 2007.

[P + 69] Alan Pritchard et al. Statistical bibliography or bibliometrics. *Journal of documentation*, 25(4):348–349, 1969.

[PD04] Parag and Pedro Domingos. Multi-relational record linkage. In Saso Dzeroski and Hendrik Blockeel, editors, *Proceedings of the 2004 ACM SIGKDD Workshop on Multi-Relational Data Mining*, pages 31–48, August 2004.

[Pen11] Si Peng. Improved matching of potential reviewers to academic papers. Masters thesis, Department of Computer Science, University of Bristol, September 2011.

[PF08] Simon Price and Peter Flach. Querying and merging heterogeneous data by approximate joins on higher-order terms. In *Inductive Logic Programming, 18th International Conference, ILP 2008, Prague, Czech Republic, September 2008, Proceedings*, volume 5194 of *LNCS/LNAI*, pages 226–243. Springer-Verlag Berlin Heidelberg, 2008.

[PF13a] Simon Price and Peter A. Flach. A higher-order data flow model for heterogeneous Big Data. In *IEEE International Conference on Big Data 2013*. IEEE Computer Society, October 2013.

[PF13b] Simon Price and Peter A. Flach. Mining and mapping the research landscape. In *Digital Research Conference*. University of Oxford, September 2013.

[PF13c] Simon Price and Peter A. Flach. A relational algebra for basic terms in a higher-order logic. Technical Report CSTR-13-004, University of Bristol, July 2013.

[PFS10a] Simon Price, Peter A. Flach, and Sebastian Spiegler. SubSift: a novel application of the vector space model to support the academic peer review process. In *Workshop on Applications of Pattern Analysis (WAPA 2010)*. Windsor, UK, September 2010.

[PFS10b] Simon Price, Peter A. Flach, and Sebastian Spiegler. SubSift: a novel application of the vector space model to support the academic research process. *Journal of Machine Learning Research - Proceedings Track*, 11:20–27, 2010.

[PFS10c] Simon Price, Peter A. Flach, and Sebastian Spiegler. SubSift: a novel application of the vector space model to support the academic research process. *Journal of Machine Learning Research - Proceedings Track*, 11:20–27, 2010.

[PFS+10d] Simon Price, Peter A. Flach, Sebastian Spiegler, Christopher Bailey, and Nikki Rogers. SubSift web services and workflows for profiling and comparing scientists and their published works. In *Proceedings of the 2010 IEEE Sixth International Conference on e-Science*, pages 182–189. IEEE Computer Society, 2010.

[PFS+13] Simon Price, Peter A. Flach, Sebastian Spiegler, Christopher Bailey, and Nikki Rogers. SubSift web services and workflows for profiling and comparing scientists and their published works. *Future Generation Comp. Syst.*, 29(2):569–581, 2013.

[PMM+03] Hanna Pasula, Bhaskara Marthi, Brian Milch, Stuart Russell, and Ilya Shpitser. Identity uncertainty and citation matching. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *Advances in Neural Information Processing 15 (NIPS 2002)*, volume 15, pages 1401–1408, Cambridge, MA, 2003. MIT Press.

[Pri03] Simon Price. A review of the state of the art of machine learning on the semantic web. In *Proceedings of 2003 UK Workshop on Computational Intelligence*, pages 292–299. University of Bristol, September 2003.

[Pri04] Simon Price. A review of the state of the art in machine learning on the semantic web. Technical Report CSTR-04-005, Department of Computer Science, University of Bristol, October 2004.

[PS05] Eric Prud'hommeaux and Andy Seabourne. *SPARQL Query Language for RDF*. W3C, W3C Working Draft 19 April 2005 edition, April 2005.

[Rae08] Luc De Raedt. *Logical and Relational Learning*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2008.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal The International Journal on Very Large Data Bases*, 10(4):334–350, December 2001.

[RC04] Pradeep Ravikumar and William W. Cohen. A hierarchical graphical model for record linkage. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 454–461, Arlington, Virginia, United States, 2004. AUAI Press.

[RCDS14] Jonathan Robie, Don Chamberlin, Michael Dyck, and John Snelson. XML Path Language (XPath) 3.0. Technical report, W3C, April 2014.

[Reu06] Patrick Reuther. Personal name matching: New test collections and a social network based approach. Technical Report 06-1, University of Trier, 54296 Trier, Germany, March 2006.

[Rit79] Dennis Ritchie. The evolution of the unix time-sharing system. In *Language Design and Programming Methodology*, pages 25–36, 1979.

[RU11] Anand Rajaraman and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[RZGSS04] Michal Rosen-Zvi, Thomas Griffiths, Mark Steyvers, and Padhraic Smyth. The author-topic model for authors and documents. In *AUAI '04: Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 487–494, Arlington, Virginia, United States, 2004. AUAI Press.

[SAYMY08] Julia Stoyanovich, Sihem Amer-Yahia, Cameron Marlow, and Cong Yu. Leveraging tagging to model user interests in del.icio.us. In *AAAI Spring Symposium: Social Information Processing*, pages 104–109. AAAI, 2008.

[SDH09] Anish Das Sarma, Xin Luna Dong, and Alon Y. Halevy. Data modeling in dataspace support platforms. In *Conceptual Modeling: Foundations and Applications*, pages 122–138, 2009.

[SDH11] Anish Das Sarma, Xin Luna Dong, and Alon Y. Halevy. Uncertainty in data integration and dataspace support platforms. In *Schema Matching and Mapping*, pages 75–108. 2011.

[Seb97] Michèle Sebag. Distance induction in first order logic. In Lavrac and Dzeroski [LD97], pages 264–272.

[SGL07] Amit P. Sheth, Karthik Gomadam, and Jon Lathem. SA-REST: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91–94, 2007.

[SHMG10] Jacek Sroka, Jan Hidders, Paolo Missier, and Carole Goble. A formal semantics for the taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490 – 508, 2010. Special Issue: Scientific Workflow 2009 The 2nd International Workshop on Workflow Management and Application in Grid Environments, The 3rd International Workshop on Workflow Management and Applications in Grid Environments.

[SJ72] Karen Spärck-Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[SKD10] Mirko Sonntag, Dimka Karastoyanova, and Ewa Deelman. Bridging the Gap between Business and Scientific Workflows: Humans in the Loop of Scientific Workflows. In *Proceedings of the 2010 IEEE Sixth International Conference on e-Science*, pages 206–213, Washington, DC, USA, 2010. IEEE Computer Society.

[soa07] Soap version 1.2 part 1: Messaging framework (second edition), April 2007.

[Sow99] John F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Course Technology, August 1999.

[Spi06] Sebastian R. Spiegler. *Comparative study of clustering algorithms on textual databases - Clustering of curricula vitae into comptency-based groups to support knowledge management*. VDM Verlag Dr. Mueller e.K., 2006.

[STC04] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, June 2004.

[SWY75a] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[SWY75b] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, 1975.

[TMG+07] D. Turi, P. Missier, C. Goble, D. De Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *e-Science and Grid Computing, IEEE International Conference on*, pages 441–448, Dec 2007.

[TMM+13] Kenjiro Taura, Takuya Matsuzaki, Makoto Miwa, Yoshikazu Kamoshida, Daisaku Yokoyama, Nan Dun, Takeshi Shibata, Choi Sung Jun, and Jun'ichi Tsujii. Design and implementation of gxp make - a workflow system based on make. *Future Generation Comp. Syst.*, 29(2):662–672, 2013.

[vdAtH05] W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.

[WA99] Barry Wilkinson and Michael Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78, New York, NY, USA, 1990. ACM.

[War63] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301):236–244, 1963.

[Whi09] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[WHW10] Paul Watson, Hugo Hiden, and Simon Woodman. e-science central for carmen: Science as a service. *Concurr. Comput. : Pract. Exper.*, 22(17):2369–2380, December 2010.

[Win88] William E. Winkler. Using the EM algorithm for weight computation in the Fellegi-Sunter model of record linkage. *American Statistical Association, Proceedings of the Section on Survey Research Methods*, pages 667–671, 1988.

[Win99] William E. Winkler. The state of record linkage and current research problems. Technical report, U. S. Bureau of the Census, Statistical Research Division, Room 3000-4, Bureau of the Census, Washington, DC, 20233-9100 USA, 1999.

[Win03] William E. Winkler. Data cleaning methods. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, August 24-27, 2003, Washington, DC*. ACM Press, August 2003.

[WKKH05] Adam Woznica, Alexandros Kalousis, Melanie Hilario Alexandros Kalousis, and Melanie Hilario. Kernels over relational algebra structures. In *PAKDD*, pages 588–598, 2005.

[YsK03] Dawit Yimam-seid and Alfred Kobsa. Expert finding systems for organizations: Problem and domain analysis and the demoir approach. *Journal of Organizational Computing and Electronic Commerce*, 13:2003, 2003.

[Zen10] Cheng Zeng. Profiling with SubSift and subgroup discovery. Masters thesis, Department of Computer Science, University of Bristol, September 2010.

[ZL04a] Dell Zhang and Wee S. Lee. Learning to integrate web taxonomies. *Web Semantics: Science, Services and Agents on the World Wide Web*, 2(2):131–151, December 2004.

[ZL04b] Hai Zhuge and Yanyan Li. Semantic profile-based document logistics for cooperative research. *Future Generation Computer Systems*, 20(1):47 – 60, 2004.